

INTRODUCCIÓN

AL

LENGUAJE

C / C++



# Introducción al lenguaje

**C / C++**

Luis Antonio Alvarez Llorente  
Juan Manuel Murillo Rodríguez  
Jorge Quirós Rosado  
Fernando Sánchez Figueroa



	<i>página</i>
PRÓLOGO. . . . .	7
 <b>CAPÍTULO 1.    <i>Introducción.</i></b>	
1.1. Orígenes y evolución. . . . .	9
1.2. Características generales. . . . .	10
1.3. Ejemplos. . . . .	10
1.4. Uso del C. . . . .	12
1.5. Consideraciones generales. . . . .	13
 <b>CAPÍTULO 2.    <i>Tipos de datos.</i></b>	
2.1. Introducción. . . . .	15
2.2. Constantes y variables. . . . .	16
2.3. Tipos de datos. . . . .	18
2.4. Constantes frente a variables.. . . .	21
2.5. Modificadores. . . . .	22
2.6. Ejemplo. . . . .	24
 <b>CAPÍTULO 3.    <i>Operadores y Expresiones.</i></b>	
— 3.1. Introducción. . . . .	25
3.2. Operadores aritméticos. . . . .	26
3.3. Operadores relacionales. . . . .	27
3.4. Operadores lógicos. . . . .	27
3.5. Operadores de bits. . . . . —	28
3.6. Operadores de asignación. . . . .	29
3.7. Operadores especiales. . . . .	31
3.8. Tipo de una expresión. . . . .	31
3.9. Precedencia y asociatividad de operadores. . . . .	32
 <b>CAPÍTULO 4.    <i>Entrada y salida de datos por pantalla.</i></b>	
4.1. Introducción. . . . .	35
4.2. Entrada y salida de caracteres. . . . .	36
4.3. Entrada y salida de cadenas. . . . .	36
4.4. Entrada y salida de datos formateada. . . . .	36

	<i>página</i>
<b>CAPÍTULO 5.    <i>Control del flujo.</i></b>	
5.1. Introducción. . . . .	39
5.2. Condiciones. . . . .	40
5.3. Instrucciones simples y compuestas. . . . .	40
5.4. Estructuras condicionales. . . . .	41
5.4.1. Estructura if....else. . . . .	41
5.4.2. El operador condicional. . . . .	42
5.4.3. Estructura switch. . . . .	43
5.5. Estructuras iterativas. . . . .	45
5.5.1. Estructura while.. . . .	45
5.5.2. Estructura do....while. . . . .	46
5.5.3. Estructura for. . . . .	47
5.6. Sentencia goto. . . . .	48
5.7. Sentencia continue. . . . .	49
<b>CAPÍTULO 6.    <i>Funciones.</i></b>	
6.1. Introducción. . . . .	51
6.2. Definición de funciones. . . . .	52
6.3. Valor devuelto por una función. . . . .	54
6.4. Declaración de funciones (prototipos). . . . .	55
6.5. La función main(). . . . .	56
6.6. Funciones de la librería estándar de C. . . . .	57
6.7. Funciones con número variable de argumentos . . . . .	57
<b>CAPÍTULO 7.    <i>Ámbito de las variables.</i></b>	
7.1. Introducción. . . . .	—61
7.2. Tiempo de vida de los objetos. . . . .	62
7.3. Especificación de tipos de almacenamiento. . . . .	62
<b>CAPÍTULO 8.    <i>Arrays.</i></b>	
8.1. Introducción. . . . .	65
8.2. Arrays unidimensionales. . . . .	66
8.3. Arrays unidimensionales como cadenas de caracteres. . . . .	68
8.4. Arrays bidimensionales. . . . .	68
8.5. Arrays multidimensionales. . . . .	70
8.6. Inicialización de arrays.. . . .	71
8.7. Ejemplo. . . . .	72

<b>CAPÍTULO 9.</b>	<b><i>Punteros.</i></b>	
9.1.	Introducción.	75
9.2.	Variables de tipo puntero y operadores.	76
9.3.	Aritmética de punteros.	78
9.4.	Punteros y funciones.	79
9.5.	Asignación dinámica de memoria.	80
9.6.	Punteros y arrays.	82
9.7.	Punteros a funciones.	84
<b>CAPÍTULO 10.</b>	<b><i>Estructuras, uniones y tipos enumerados.</i></b>	
10.1.	Introducción.	87
10.2.	Estructuras.	88
10.2.1.	Declaración de estructuras.	88
10.2.2.	Definición de estructuras.	89
10.2.3.	Operaciones sobre estructuras.	90
10.3.	Uniones.	94
10.4.	Tipos enumerados.	95
<b>CAPÍTULO 11.</b>	<b><i>Entrada / Salida en C. Sistema de ficheros.</i></b>	
11.1.	Introducción.	99
11.2.	Sistema de E/S.	100
11.3.	Flujos y archivos.	100
11.4.	E/S por consola.	101
11.5.	El sistema de archivos ANSI C.	102
11.6.	Rutinas de archivos tipo UNIX.	105
11.7.	Ejemplos.—	107
<b>CAPÍTULO 12.</b>	<b><i>Un ejemplo de programa en C. —</i></b>	
12.1.	Introducción.	109
12.2.	Presentación del problema.	110
12.3.	El T.A.D.	110
12.4.	El código.	113
12.5.	Conclusión	121
<b>CAPÍTULO 13.</b>	<b><i>El preprocesador y las opciones de compilación.</i></b>	
13.1.	Introducción.	123
13.2.	Preprocesado. Directivas de compilación.	124
13.3.	Directiva de inclusión.	124
13.4.	Directivas de definición.	125
13.5.	Directivas de depuración.	130
13.6.	Directivas condicionales.	131

	página
<b>CAPÍTULO 14. <i>Introducción a C++: programación orientada a objetos.</i></b>	
14.1. Introducción. . . . .	135
14.2. Conceptos fundamentales. . . . .	136
14.2.1. Clases y objetos. . . . .	136
14.2.2. Paso de mensajes. . . . .	136
14.2.3. Herencia. . . . .	137
14.2.4. Redefinición de métodos. . . . .	138
14.2.5. Ligadura temprana vs ligadura tardía. . . . .	138
14.2.6. Generalidad. . . . .	139
14.3. Clases en C++. . . . .	140
14.3.1. Definición de Clases . . . . .	140
14.3.2. Instanciación de clases: Objetos. . . . .	141
14.4. Constructores y destructores. . . . .	141
14.4.1. Constructores. . . . .	141
14.4.2. Destructores . . . . .	142
14.4.3. Constructores y destructores por defecto. . . . .	142
14.4.4. Argumentos por omisión . . . . .	143
14.5. Herencia en C++. . . . .	143
14.5.1. Expresión de la herencia en C++ . . . . .	143
14.6. Polimorfismo y sobrecarga de funciones. . . . .	145
14.6.1. Polimorfismo . . . . .	145
14.6.2. Sobrecarga . . . . .	146
14.7. Uso de templates. . . . .	147
<b>CAPÍTULO 15. <i>Un ejemplo de programa en C++.</i></b>	
15.1. Introducción. . . . .	149
15.2. El código. . . . .	150
<b>APÉNDICE A. <i>Modelos de memoria.</i></b> . . . . .	159
<b>APÉNDICE B. <i>Librerías estándar del C..</i></b> . . . . .	165
<b>APÉNDICE C. <i>Ficheros Proyecto. Construcción de grandes aplicaciones.</i></b> . . . . .	179
<b>APÉNDICE D. <i>Bibliografía recomendada.</i></b> . . . . .	187



El objetivo fundamental de este libro es introducir al interesado en la programación en C. Como cuestión primordial, los autores plantean el aprendizaje gradual, de manera que se pueda comenzar a escribir pequeños programas en C desde el primer día.

Aunque los ejemplos descritos en las páginas que vienen a continuación han sido codificados en un entorno de programación concreto, Borland, sobre un sistema operativo específico D.O.S, los conceptos que aquí se plantean pueden aplicarse a cualquier entorno de programación C, tanto en aquellos basados en Windows como en entornos UNIX.

Desde mi punto de vista, este libro no debe concebirse como una enciclopedia del C, pues ese tratamiento restringiría el contenido del libro a un entorno de programación concreto al tener que utilizar bibliotecas de funciones específicas. Por el contrario, contiene lo necesario para que el lector adquiriera los conocimientos suficientes para poder desenvolverse posteriormente en cualquier entorno de programación C.

El libro contiene aquello que nos gustaría que dominaran la mayoría de los alumnos al finalizar un curso de programación en C. Desde los tipos de datos básicos que aporta el lenguaje y que permite el desarrollo de aplicaciones utilizando una programación estructurada y modular, hasta el concepto de clase que aporta la programación orientada a objetos en C++.

En este sentido, el capítulo dedicado a la programación en C++ debe entenderse como una introducción a los conceptos básicos de la programación orientada a objetos en general, centrando las características básicas que aportan los lenguajes orientados a objetos al ámbito particular de C++.

El libro está dirigido a alumnos de un primer curso de programación en C y a cualquier persona que interesada en introducirse en la programación en C.



# CAPÍTULO 1

---

## *Introducción*

### 1.1. Orígenes y evolución.-

La evolución de C está muy ligada a la del UNIX. Por tanto el éxito de uno ha intervenido directamente en el éxito del otro.

En 1969, Ken Thompson iniciaría el desarrollo del sistema operativo UNIX, sobre una máquina PDP-7 de Digital. El lenguaje utilizado para el desarrollo del sistema fue el B, un lenguaje experimental que estaba siendo desarrollado en la misma época por Martin Richards y que estaba muy cerca del ensamblador. El nuevo sistema operativo era lo suficientemente atractivo como para justificar la adquisición de una máquina de mayores prestaciones: la PDP-11. Uno de sus primeros usuarios fue Dennis Ritchie, que la usó para desarrollar el lenguaje C, como extensión del B, entre 1970 y 1972. Más tarde, Thompson y Ritchie usan el lenguaje C para reescribir la mayor parte del sistema operativo UNIX. De esta forma, C se convirtió en el principal lenguaje de desarrollo de software de sistemas.

En 1978, Brian Kernighan colaboró con Dennis Ritchie en la redacción del primer manual de referencia de C, *The C Programming Language*.

Posteriormente, la organización internacional ANSI publicó un documento de descripción normalizada de C. Esta versión incorporó algunas modificaciones con respecto al C de K&R, pero conservó casi todas sus características. Actualmente, la mayoría de los compiladores de C han adoptado en mayor o menor medida las especificaciones dadas por el estándar ANSI.

## 1.2. Características generales.-

Se pueden citar algunas que hacen del C un lenguaje muy usado:

- *Es un lenguaje de propósito general.* Este lenguaje se ha utilizado para el desarrollo de aplicaciones tan dispares como: hojas de cálculo, gestores de bases de datos, sistemas operativos, compiladores, software de gestión, comunicaciones, etc.

- *Es un lenguaje de medio nivel.* Tiene las prestaciones de un lenguaje de alto nivel sin dejar de lado las de bajo nivel (máxima eficiencia y control absoluto de cuanto sucede en el interior del ordenador).

- *Es portable.* Los programas escritos en C son fácilmente portables de un sistema a otro.

- *Es potente y eficiente.* Usando C, un programador puede casi alcanzar la eficiencia del código ensamblador junto con la estructura del Pascal.

Es un lenguaje fácil de aprender, sólo tiene 32 palabras claves (27 del estándar de Kernighan y Ritchie y 5 añadidas por el comité de estandarización del ANSI) pero para sacar un rendimiento óptimo de su uso hay que tener una gran experiencia. Como se suele decir, a programar se aprende programando. Así es que antes de nada veremos algunos ejemplos de programas en C y sobre ellos veremos algunos conceptos.

## 1.3. Ejemplos.-

### Ejemplo 1

```
#include <stdio.h>
main ()
{
    printf ("Mi primer programa en C.");
}
```

La salida para este programa sería: *Mi primer programa en C.*

Los programas en C se componen de unidades de programa llamadas **funciones**. Se podría decir que un programa en C es un conjunto de funciones. En nuestro ejemplo sólo hay una función y se llama **main**. Todos los programas en C deben contener al menos una función que se llame *main*. Esta será la función principal, por donde el programa comience a ejecutarse.

Los paréntesis que siguen a *main* son los que indican que es una función. Dentro de los paréntesis podemos especificar los **parámetros** que queramos. En este caso no los hay, pero los paréntesis son obligatorios. El comienzo de la función queda definido por una llave abierta ({} y el final por una llave cerrada (}) de igual forma que en PASCAL se usan el *begin* y el *end*.

La línea:

```
printf ("Mi primer programa en C.");
```

realiza una llamada a la función **printf()** y como parámetro le pasa la cadena "*Mi primer programa en C*"; **printf()** es una función de la **librería de funciones estándar del C** que realiza una escritura en la salida estándar (normalmente el monitor). La llamada a la función termina con ';'. Cada instrucción en C termina siempre con ';'.

Hemos dicho que **printf()** es una función de librería. ¿Qué significa esto? Bien, el C es un lenguaje con pocas palabras claves, pero sin embargo muy potente. Esto se consigue gracias a la incorporación al lenguaje de unas librerías de funciones disponibles para el programador. Para poder usar cualquiera de ellas es necesario incluir el fichero donde se encuentra declarada. Concretamente, **printf()** está declarada en el fichero **stdio.h**, con lo cual tendremos que incluir este fichero en nuestro programa. Esto se hace con la primera línea. En el fichero *stdio.h* se encuentran las declaraciones de las funciones y tipos de datos relacionados con la entrada/salida estándar. Existen otros ficheros de este tipo que ya veremos más adelante. El apéndice C está enteramente dedicado a las librerías estándar.—

## Ejemplo 2

```
#include <stdio.h>
main ()
{
    printf ("Mi segundo programa en C.\n");
    printf ("Pulsa la tecla RETURN para continuar.");
    getchar();
}
```

La salida por pantalla de este programa sería:

```
Mi segundo programa en C.
Pulsa la tecla RETURN para continuar.
```

Hay 2 novedades con respecto al primer ejemplo: la aparición del carácter `\n` y la aparición de la función de librería **getchar()**.

El carácter `\n` es un carácter especial que denota nueva línea. Al ser un carácter puede incluirse en cualquier cadena, como en nuestro ejemplo. Por eso en la salida aparece la cadena del segundo `printf()` en otra línea.

`getchar()` es una función de librería que se encuentra en `<stdio.h>`, por lo que es necesario incluir este fichero al principio del programa. Esta función espera la pulsación de la tecla return. No necesita pasar ningún argumento, sin embargo los paréntesis son necesarios.

### Ejemplo 3

```
#include <stdio.h>
main () /* tercer ejemplo */
{
    int horas, minutos;

    horas = 3;
    minutos = 60*horas;
    printf ("Hay %d minutos en %d horas.", minutos, horas);
    getchar ();
}
```

De nuevo aparecen cosas nuevas en este ejemplo:

Cualquier cosa que en C aparezca encerrada entre `/*...*/` es un comentario.

Con la línea `int horas, minutos`, se están declarando 2 variables de tipo entero, cuyos identificadores son *horas* y *minutos*. Al final tiene que ir un `;` al igual que al final de cada instrucción.

La sentencia `printf` escribe:

*Hay 180 minutos en 3 horas.*

Como se ve, los dos `%d` no se han escrito y en su lugar han aparecido los valores almacenados en las variables *minutos* y *horas*.

El símbolo `%d` indica a la función `printf` que lo que se tiene que escribir a continuación es una variable. Esta variable tendrá que venir también como argumento en la propia función. La *d* le indica a `printf` que lo que tiene que imprimir es una variable entera.

No hay que preocuparse si no se ha entendido algo de estos 3 ejemplos. Todo lo que hay en ellos será explicado con más detalles en los capítulos siguientes. Con estos ejemplos lo que se ha pretendido es hacer programas completos en C desde el principio, intentado ofrecer una visión global del mismo.

1.4. Uso del C.-

Los pasos que se siguen desde que se comienza a escribir un programa en C hasta que se ejecuta, son los siguientes:

- **Escribirlo en un editor:** Cualquier editor es válido siempre y cuando genere ficheros de texto estándar, es decir, sin caracteres de control ni caracteres no imprimibles. Normalmente un programa en C ocupará varios ficheros.
- **Compilarlo:** El compilador (*compiler*) produce ficheros objeto, con extensión **.OBJ**. Un fichero objeto es aquel que contiene instrucciones en código máquina. Serán utilizados como entrada al enlazador.
- **Enlazarlo:** El enlazador (*linker*) se encarga de coger todos los ficheros objeto que forman un programa y combinarlos para formar un fichero directamente ejecutable desde el Sistema Operativo. También hay ficheros con extensión **.LIB** (ficheros de librería) que también pueden ser combinados con los **.OBJ** para formar el ejecutable.
- **Ejecutarlo:** Se ejecuta normalmente, tecleando su nombre y posibles argumentos desde la línea de órdenes del sistema operativo que se esté usando.

Un esquema de los pasos expuestos puede ser el que se presenta en la figura 1.1.

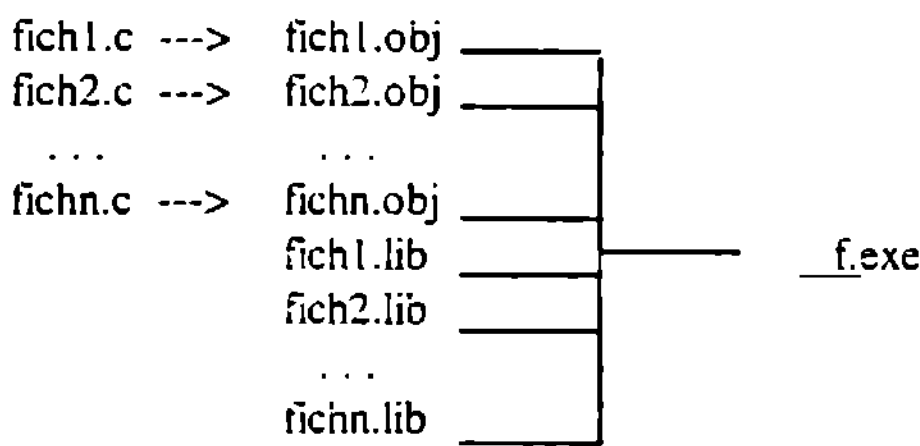


Figura 1.1 Esquema de pasos para ejecución de un programa en C.

1.5. Consideraciones generales.-

Hoy en día, los compiladores de C son muy sofisticados e incluyen entornos integrados desde los que se pueden hacer los 4 pasos descritos anteriormente. Nosotros a la hora de redactar este libro nos hemos basado en un compilador bastante potente de la casa **Borland** que es el **Turbo C**. No obstante, todo lo explicado en sucesivos capítulos es válido para cualquier compilador de C, pues está basado en el **ANSI C**. En aquellos sitios donde se use algo específico del **Turbo C** se indicará oportunamente.





## *Tipos de datos*

### 2.1. Introducción.-

El objetivo de este capítulo es el de presentar los distintos tipos de datos que permite utilizar el C. En posteriores capítulos veremos dónde almacenar y cómo manipular esos datos.

Una primera división de los datos podríamos hacerla en **constantes y variables**, dependiendo de si su valor puede o no variar a lo largo de la ejecución del programa.

Pero tanto las constantes como las variables han de ser de algún tipo. Y precisamente esta es otra de las grandes divisiones que podemos hacer con respecto a los datos: **su tipo**. Veremos los diferentes tipos de datos que soporta el C y las relaciones que se pueden establecer entre datos de diferentes tipos.

## 2.2. Constantes y variables.-

Las constantes son datos con valores fijos que no pueden ser alterados por el programa, mientras que las variables son datos cuyo valor puede cambiar a lo largo del programa.

Antes de pasar a ver los distintos tipos de datos, veamos primero cómo se declaran y asignan las variables en C, pues será necesario para comprender algunos ejemplos posteriores.

### Declaración de variables.-

La declaración de variables es necesaria puesto que el compilador antes de usar una variable debe conocer en primer lugar su nombre, y en segundo lugar el tipo de datos al que pertenece. De esta forma se puede determinar sus necesidades de almacenamiento (cuánto ocupa) y se puede comprobar en qué contextos es válido utilizar la variable (ámbito).

La sintaxis para la declaración de variables es:

```
tipo identificador [, identificador ...];
```

Como se ve, se puede declarar más de una variable en la misma línea. Dado que las variables deben ser declaradas antes de utilizarlas, el lugar apropiado es al comienzo del cuerpo de la función.

Un **identificador** consiste en una secuencia continua (sin espacios en blanco) de letras (A..Z,a..z), carácter de subrayado (`_`) y dígitos (0..9), que empieza con una letra o un carácter de subrayado, y que no es idéntico a ninguna palabra clave del C.

Hay que destacar que el C es *case sensitive*, es decir, distingue entre mayúsculas y minúsculas. Por tanto, no será lo mismo el identificador *dni* que el identificador *DNI*.

Las palabras claves del C son:

---

<i>auto</i>	<i>break</i>	<i>case</i>	<i>char</i>
<i>const</i>	<i>continue</i>	<i>default</i>	<i>do</i>
<i>double</i>	<i>else</i>	<i>enum</i>	<i>extern</i>
<i>float</i>	<i>for</i>	<i>goto</i>	<i>if</i>
<i>int</i>	<i>long</i>	<i>register</i>	<i>return</i>
<i>short</i>	<i>signed</i>	<i>sizeof</i>	<i>static</i>
<i>struct</i>	<i>switch</i>	<i>typedef</i>	<i>union</i>
<i>unsigned</i>	<i>void</i>	<i>volatile</i>	<i>while</i>

---

Ejemplo de declaraciones válidas:

```
int radio, centro;  
char uncharacter;  
long int alfa, beta;  
double precision;
```

### Asignación de variables.-

Una vez declarada, la primera operación que se debe hacer con una variable es asignarle un valor inicial. El compilador no inicializa las variables, es decir, no les asigna ningún valor por defecto, y además no detecta error si se intenta utilizar una variable no inicializada.

La sintaxis de la asignación es:

*variable = valor;*

donde valor puede ser:

- una constante
- otra variable
- el valor devuelto por una función
- una expresión, formada por una combinación de los anteriores mediante operadores.

Ejemplos:

```
x = 3-y;  
x = abs(-5);  
x = y;  
x = abs(y) + floor(z) + 1;
```

Es posible realizar declaración e inicialización en una sola instrucción:

```
int total = 0;  
float pi = 3.14159;  
int suma = total + 1; /* sólo válida si total ya ha sido inicializada */
```

Ya veremos en el próximo capítulo qué ocurre cuando en las asignaciones el valor de la parte derecha no es del mismo tipo que la variable de la parte izquierda.

2.3. Tipos de datos.-

En C existen 5 tipos básicos de datos:

Tipo	Descripción	Longitud	Rango
char	carácter	1	0 a 255
int	entero		-32768 a 32767
float	coma flotante	4	6 dígitos de precisión (aprox)
double	coma flotante	8	12 dígit. de precisión (aprox.)
void	sin valor	0	sin valor

Hay que tener en cuenta que la longitud (viene dada en bytes) y el rango, dependen del compilador y la máquina que se esté usando. No obstante la información contenida en la tabla es válida para la mayoría de los compiladores.

Tipo Carácter.-

Un carácter se representa delimitado por comillas simples, también llamados apóstrofes.

Ejemplos válidos: 'a', '1', '&'. . .

Ejemplos inválidos: 'cc', r, 6

'cc' es incorrecto pues hay 2 caracteres entre los apóstrofes.

r es incorrecto pues es interpretado como una variable.

6 es incorrecto pues es interpretado como una constantes entera.

El valor de una constante carácter es el valor numérico del carácter en el conjunto de caracteres que use el sistema. Por ejemplo, en el conjunto ASCII, el carácter '0' es 48, mientras que en EBCDIC es 240. Nos referiremos normalmente al conjunto de caracteres ASCII.

Existen representaciones para algunos caracteres no imprimibles o especiales:

Carácter	Descripción
\a	Campana
\b	Retroceso
\f	Salto de página
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador horizontal
\v	Tabulador vertical
\\	Barra invertida
\"	Comillas (")
\'	Apóstrofe (')

Para representar otros caracteres especiales que no aparecen en esta tabla se puede usar su código ASCII o bien:

<code>\nnn</code>	Representa el carácter de código octal <i>nnn</i>
<code>\xnn</code>	Representa el carácter de código hexadecimal <i>nn</i>

Ejemplo de asignaciones equivalentes:

```
char ch1, ch2, ch3, ch4; /* declara 4 variables tipo carácter */
ch1 = '\n';
ch2 = 13; /* el carácter '\n' es el número 13 en ASCII */
ch3 = '\15' /* 13 en decimal es 15 en octal */
ch4 = '\xD' /* 13 en decimal es D en hexadecimal */
```

Hay que tener en cuenta que las 3 últimas notaciones vistas son dependientes del código que se esté utilizando (en nuestro caso el ASCII) y por tanto debieran ser evitadas en lo posible por el tema de la portabilidad, aunque habrá ocasiones en las que no habrá más remedio que usarlas.

### Tipo entero.-

Es un número que no tiene parte fraccionaria. Se pueden escribir de uno de los 3 modos siguientes:

- En decimal: escribiendo el número sin empezar por 0 (a excepción de que sea el propio 0). Ejemplos: 2, 1, 0, -2.
- En octal: empezando el número por 0. Ejemplos: 02, 07, 020.
- En hexadecimal: empezando el número por 0x. Ejemplos: 0xE, 0x1d, 0x4.

### Tipos float y double.-

Las constantes de este tipo tienen parte real y parte fraccionaria. La diferencia entre ellos es que el tipo float tiene la mitad de precisión que el tipo double. La sintaxis correcta de ambos tipos es:

*[signo] [dígitos] [.] [dígitos] [exponente [signo] dígitos]*

donde:

*signo* es + ó -

*dígitos* es una secuencia de dígitos

*.* es el punto decimal

*exponente* es E ó e

Los elementos entre corchetes son opcionales. El número no puede empezar directamente por e ó E. Ejemplos válidos : 1.2e9, -3E-8, -2.5

## Tipo void.-

Significa sin valor, sin tipo. El uso clásico de void puede apreciarse al comparar estos 2 programas que hacen exactamente lo mismo:

```
#include <stdio.h>
main ()
{ printf ("Versión 1"); }
```

```
#include <stdio.h>
void main (void)
{ printf ("Versión 2"); }
```

Al poner *void* entre los paréntesis de la definición se indica que la función no tiene parámetros. Al poner *void* antes del nombre de la función en la definición de ésta, se está declarando como función que no devuelve nada. Es conveniente utilizar la segunda versión.

## Otros tipos.-

### Tipos array, puntero y estructura.-

Por la importancia de estos tipos se les dedica un capítulo a cada uno de ellos. Básicamente:

- un array es una colección de elementos del mismo tipo.
- un puntero es la dirección de otro objeto.
- una estructura es una colección de elementos que pueden ser de distinto tipo.

## Tipos definidos.-

La palabra clave **typedef** crea un tipo definido. Por ejemplo :

```
typedef int BOOLEAN;
```

hace un nuevo tipo *BOOLEAN*, equivalente al entero. La declaración

```
BOOLEAN quit;
```

es equivalente a la declaración de la variable *quit* como de tipo entero.

## El tipo enum.-

La palabra clave **enum** crea una lista de enteros constantes que pueden ser asignados a un objeto específico. Por ejemplo:

```
enum BOOLEAN (FALSE, TRUE) quit;
enum BOOLEAN proceed;
```

especifica que las variables *quit* y *proceed* son del tipo entero, a las que se les puede asignar los valores enumerados FALSE ó TRUE.

## 2.4. Constantes frente a variables.-

Las constantes son valores fijos que no pueden ser alterados por el programa. En C se pueden utilizar los siguientes tipos de constantes:

- caracteres
- números decimales
- números hexadecimales
- números octales
- cadenas de caracteres
- constantes definidas

Las variables por su parte sí que pueden modificar su valor durante la ejecución del programa. Antes de utilizar cualquier variable será necesario proceder a su declaración.

Los tipos asociados a las 4 primeras ya se han comentado en apartados anteriores. Sólo añadir que las constantes enteras se representan internamente por defecto como pertenecientes al tipo *int*. Pero también es posible indicar otro tipo de almacenamiento añadiendo al final de la constante un sufijo indicándolo. Por ejemplo:

123	<i>int</i>
123L	<i>long</i>
123U	<i>unsigned int</i>
123UL	<i>unsigned long</i>

En la siguiente sección veremos qué significa eso de *unsigned* y *long*.

### Constantes de tipo cadena.-

Una constante tipo cadena es una secuencia de caracteres delimitada por comillas dobles (""). Veámoslo con un ejemplo:

```
char respuesta[] = ("Ganó 4 a 0.");
puts ("¿Cómo quedó el Real Madrid el domingo?\n");
puts (respuesta);
```

se produce la salida:

```
¿Cómo quedó el Real Madrid el domingo?
Ganó 4 a 0.
```

Se pueden distinguir 2 tipos de cadenas en este ejemplo:

1. Por un lado las cadenas "*Ganó 4 a 0*" y "*¿Cómo quedó el Real Madrid el domingo?*" que son constantes, no ocupan memoria direccionable en los segmentos de datos del programa y por tanto son fijas e invariables.

2. Por otro lado tenemos *respuesta*, que es una variable de tipo cadena, que sí ocupa memoria direccionable y por tanto es susceptible de modificación. Ya veremos en el capítulo 8 cómo se declaran y manipulan las variables de tipo cadena.

Constantes definidas.-

La directiva del compilador `#define` permite definir identificadores para representar constantes. Por ejemplo:

```
#define NUMEROMAX 25
#define ESC 0x1B
#define PI 3.14159
#define AVOGADRO 6.023E+23
#define RESPUESTA 's'
#define SALUDO "Hola, amigos"
```

Estas constantes se evalúan en tiempo de compilación. El compilador cada vez que se encuentre la palabra *SALUDO*, la sustituirá por su valor *"Hola, amigos"*. El uso de esta directiva está ampliamente comentado en el apéndice A.

2.5. Modificadores.-

Modificadores de Tipo

Existen unos **modificadores de tipo** que sirven para ajustar más los datos a las necesidades del tratamiento y/o almacenamiento. Estos modificadores son:

Modificador	Descripción	Tipos a los que se aplican
signed	con signo	int, char
unsigned	sin signo	int, char
long	largo	int, char, double
short	corto	int, char

El uso de *signed* con enteros es redundante, aunque está permitido.

Se puede utilizar un modificador de tipo sin tipo; en este caso, el tipo se asume que es *int*.

La longitud de los tipos depende del sistema que utilicemos. No obstante, la siguiente tabla es válida para la mayoría de los sistemas.



Tipo	Longitud	Rango
char	1	Caracteres ASCII
unsigned char	1	0 a 255
signed char	1	-128 a 127
int	2	-32768 a 32767
unsigned int	2	0 a 65535
signed int	2	Igual que int
short int	1	-128 a 127
unsigned short int	1	0 a 255
signed short int	1	Igual que short int
long int	4	-2147483648 a 2147483647
signed long int	4	Igual que long int
unsigned long int	4	0 a 4294967296
float	4	6 dígitos de precisión (aprox.)
double	8	12 dígitos de precisión (aprox.)
long double	16	24 dígitos de precisión (aprox.)
void	0	sin valor

Modificadores de acceso.-

Sirven para modificar el acceso a los tipos. Son dos: **const** y **volatile**.

Las variables de tipo **const** son aquellas a las que se les asigna un valor inicial y este valor no puede alterarse a lo largo del programa. Un ejemplo de cómo se declara:

```
---
const unsigned int hola;
```

Por su parte, las variables de tipo **volatile** indican al compilador que esa variable puede alterar su valor por medios no especificados explícitamente en el programa.

2.6. Ejemplo.-

Programa que muestra el tamaño de los tipos más usados en C en el sistema en el que se ejecute el programa.

```
#include <stdio.h> /* printf(), getch() */

void main (void)
{
    printf ("TAMAÑO DE ALGUNOS TIPOS :\n\n");
    printf ("      Tipo      Tamaño (en bytes)\n");
    printf ("-----\n");
    printf ("char          %d\n", sizeof (char));
    printf ("int           %d\n", sizeof (int));
    printf ("short int     %d\n", sizeof (short int));
    printf ("long int      %d\n", sizeof (long int));
    printf ("float         %d\n", sizeof (float));
    printf ("double        %d\n", sizeof (double));
    printf ("long double   %d\n", sizeof (long double));
    printf ("\nPulsa la tecla RETURN para salir.\n");
    getch ();
}
```

Este programa dará como salida :

TAMAÑO DE ALGUNOS TIPOS :

Tipo	Tamaño (en bytes)
-----	-----
char	1
int	2
short int	1
long int	4
float	4
double	8
long double	16

Pulsa la tecla RETURN para salir.

## *Operadores y expresiones*

### 3.1. Introducción.-

Una característica muy importante del lenguaje de programación C es la cantidad y variedad de operadores que posee. Operadores y operandos se mezclan para formar expresiones. Cuando en una expresión se usa más de un operador, el compilador aplicará las reglas de prioridad para decidir qué operación ha de hacerse primero en la evaluación de la expresión completa. No obstante, el uso de paréntesis puede variar el orden de evaluación a nuestro antojo.

El lenguaje C ofrece más de 40 operadores distintos que pasaremos a ver en este capítulo, y que pueden agruparse en 6 grupos diferentes:

- ◆ operadores aritméticos
- ◆ operadores relacionales
- ◆ operadores lógicos
- ◆ operadores de bits
- ◆ operadores de asignación
- ◆ operadores especiales

### 3.2. Operadores aritméticos.-

La tabla 3.1 refleja los operadores aritméticos que posee el C.

Operador	Tipo	Función
+	Unario	Signo positivo
+	Binario	Suma
-	Unario	Signo negativo
-	Binario	Resta
*	Binario	Multiplicación
/	Binario	División
%	Binario	Módulo o resto

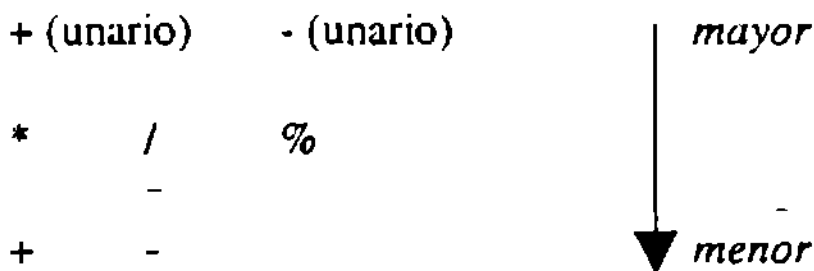
Tabla 3.1 Operadores aritméticos del C.

En C no existe operador para el cálculo de la potencia, sin embargo ya veremos que existe una función de la librería que sí lo hace.

El operador % sólo admite operandos enteros.

Los operandos deberán ser del mismo tipo. En caso de que no lo sean se aplicarán unas reglas de conversión de tipos explicadas en la última parte de este capítulo.

La precedencia (o reglas de prioridad) de los operadores aritméticos de mayor a menor es:



3.3. Operadores relacionales.-

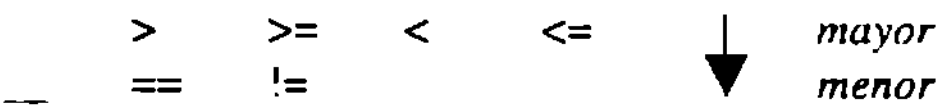
La palabra relacional se refiere a si se cumple una determinada relación entre 2 valores cualesquiera del mismo tipo. Están basadas en los valores booleanos *verdadero* y *falso*, es decir, si se cumple la relación o no se cumple. En C, cualquier cosa que sea 0 es falso, mientras que cualquier cosa distinta de 0 es verdadero.

Los operadores son los mostrados en la tabla 3.2.

Operador	Tipo	Función
>	Binario	Mayor
>=	Binario	Mayor o igual
<	Binario	Menor
<=	Binario	Menor o igual
==	Binario	Igual
!=	Binario	Distinto

Tabla 3.2.- Operadores relacionales.

La precedencia de estos operadores es de mayor a menor:



3.4. Operadores lógicos.-

Son los que pueden apreciarse en la tabla 3.3.

Operador	Tipo	Función
&&	Binario	AND
	Binario	OR
!	Unario	NOT

Tabla 3.3.- Operadores lógicos.

Los operadores se evalúan de la siguiente forma:

- A&&B
- Es verdadero sólo si A y B son verdaderos; falso en cualquier otro caso.
- A||B
- Es falso sólo si A y B son falsos; verdadero en cualquier otro caso.
- !A
- Es verdadero si A es falso, y falso si A es verdadero.

Una ventaja del lenguaje C sobre otros es que con los operadores lógicos podemos concatenar no sólo operaciones relacionales sino de cualquier tipo, ya que hemos dicho anteriormente que cualquier valor distinto de 0 es verdadero.

3.5. Operadores de bits.-

Estos operadores realizan operaciones sobre los bits de un byte o una palabra (dos bytes). Sólo se pueden usar con los tipos *char* e *int*. Son los que se presentan en la tabla 3.4.

Operador	Tipo	Función
&	Binario	AND bit a bit
	Binario	OR bit a bit
^	Binario	XOR bit a bit
<<	Binario	Desplazamiento a la izqda.
>>	Binario	Desplazamiento a la dcha.
-	Unario	Complemento a 1

Tabla 3.4.- Operadores de bits.

Hay que significar que los operadores relacionales y lógicos siempre producen un resultado que es 0 ó 1, mientras que las operaciones entre bits producen cualquier valor arbitrario de acuerdo con la operación específica.

Ejemplo:

```
char x, y, z1, z2;
x = 2; y = 3;
z1 = 2 && 3;
z2 = 2 & 3;
/* z1 = 1; z2 = 2 */
```

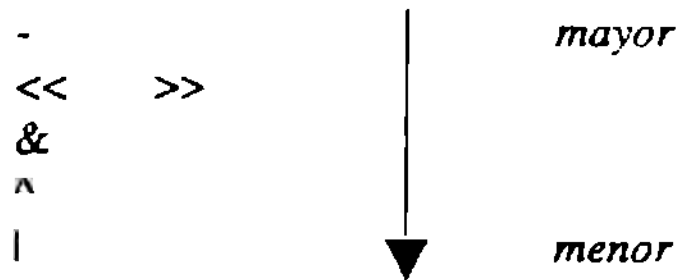
- 2 && 3 : el compilador evalúa CIERTO && CIERTO, y devuelve 1 (cierto).
- 2 & 3 : el compilador evalúa 00000010 & 00000011, que es: 00000010 (2 en decimal)

La sintaxis para los operadores de desplazamiento es:

```
expresión >> número de bits a desplazar a la derecha
expresión << número de bits a desplazar a la izquierda
```

Una observación importante sobre los operadores de desplazamiento es que un desplazamiento no es una rotación, es decir, los bits que salen por un extremo no se introducen por el otro.

El orden de precedencia de estos operadores de mayor a menor es:



Ejemplos:

```
void main (void)
{
char x, y;
/* Asignaciones
```

/* -----	x en bits	y en bits	*/
/* -----	-----	-----	*/
x = 2; y = 3;	/* 0000 0010	0000 0011	*/
y = y << 1;	/* 0000 0010	0000 0110	*/
y = x   9;	/* 0000 0010	0000 1011	*/
y = y << 3;	/* 0000 0010	0101 1000	*/
x = ~x;	/* 1111 1111	0101 1000	*/
x = 4 ^ 5 & 6;	/* 0000 0000	0101 1000	*/

```
}
```

### 3.6. Operadores de asignación.-

La asignación en C es realmente un operador como cualquier otro, con la peculiaridad de que tiene el efecto secundario de asignar un valor a una variable. Esto significa que la asignación sigue las reglas generales de todas las expresiones, y que por tanto puede usarse para formar nuevas expresiones. Por ejemplo, serían expresiones válidas las siguientes:

```
a = 1;
a = b = c = 0;          /* Asigna 0 a c, b y a */
x = (a = b) + 5;         /* Asigna b a a, y a+5 a x */
```

El valor devuelto por una expresión de asignación es el valor asignado.

El operador de asignación se evalúa de derecha a izquierda. En el ejemplo anterior, el 0 se asigna primero a c, luego a b y por último a a, y el valor de la expresión será 0.

Además de la asignación simple, en C se pueden usar otros dos tipos de asignación:

- autoincremento y autodecremento
- asignación con realimentación

Autoincremento y autodecremento.-

Para incrementar o decrementar el valor de una variable en una unidad se pueden utilizar los operadores ++ y -- de la siguiente manera:

<i>++variable</i>	Preincremento
<i>variable++</i>	Postincremento
<i>--variable</i>	Predecremento
<i>variable--</i>	Postdecremento

Como se puede apreciar, los operadores de incremento y decremento pueden preceder o seguir al operando. Si el operador precede al operando, C lleva a cabo la operación antes de utilizar el valor del operando. Si el operador sigue al operando, C utilizará su valor antes de incrementarlo o decrementarlo. Veámoslo con un par de ejemplos, pues esto suele llevar algunas veces a equívocos.

```
1      int x, y;          int x, y;
      x = 2;             x = 2;
      y = ++x;           y = x++;
      /* x = 3 e y = 3 */ /* x = 3 e y = 2 */
```

Asignación con realimentación.-

Una asignación con realimentación es aquella en la que el valor de una variable se utiliza para calcular su nuevo valor, es decir, una asignación de la forma:

*variable = variable op expresion*                      C permite abreviarlo :  
  
*variable op= expresion*

— Así se obtiene la tabla 3.5 con operadores de asignación.

Sentencia de asignación	Sentencia de asignación equivalente
<i>x *= y;</i>	<i>x = x * y;</i>
<i>x /= y;</i>	<i>x = x / y;</i>
<i>x %= y;</i>	<i>x = x % y;</i>
<i>x += y;</i>	<i>x = x + y;</i>
<i>x -= y;</i>	<i>x = x - y;</i>
<i>x &lt;&lt;= y;</i>	<i>x = x &lt;&lt; y;</i>
<i>x &gt;&gt;= y;</i>	<i>x = x &gt;&gt; y;</i>
<i>x &amp;= y;</i>	<i>x = x &amp; y;</i>
<i>x ^= y;</i>	<i>x = x ^ y;</i>
<i>x  = y;</i>	<i>x = x   y;</i>

Tabla 3.5.- Operadores de asignación con realimentación.



3.7. Operadores especiales.-

Otros operadores en C son los que se muestran en la tabla 3.6. En este capítulo no hablaremos de ellos, puesto que serán abordados con más profundidad en los siguiente capítulos.

Operador	Contexto	Función
[	Arrays	Indica rango de un array o para hacer referencia a un elemento del mismo.
*	Punteros	Devuelve el contenido de un puntero.
&	Punteros	Devuelve la dirección de un objeto.
sizeof	Cualquiera	Devuelve el tamaño de un objeto
()	Funciones	Indica que un objeto es de tipo función
.	Estructuras	Hace referencia a un campo de una estructura o unión
->	Estructuras	Hace referencia a un campo de una estructura o unión a través de un puntero
?:	Cualquiera	Evalúa una expresión en función de una condición
,	Bucle for, etc	Permite concatenar varias expresiones

Tabla 3.6.- Operadores especiales.

3.8. Tipo de una expresión.-

Antes de evaluar una expresión binaria es necesario que ambos operandos pertenezcan al mismo tipo de datos. En caso de que la expresión combine distintos tipos de datos, debe producirse una conversión de tipo antes de poder evaluarla.

Cuando hay diferencias de tipo en una *asignación*, la regla de conversión es simple: el valor del lado derecho de la expresión se convierte al tipo de la del lado izquierdo.

Ejemplo:

```
int x = 2.3; /* 2.3 se convierte a 2 */
char ch = 500; /* los bits más significativos de 500 se pierden */
```

Cuando tenemos un *operador* con dos operandos de tipo diferente, el tipo de la expresión resultante es el de mayor tamaño (mayor longitud en bytes). Consideremos el siguiente ejemplo:

```
int i;
char c;
double d;
d = d + i * c;
```

Primero se lleva a cabo la multiplicación, dado que tiene mayor precedencia. El valor de *c* se convierte a tipo *int*, y el resultado de la suma es de tipo *int*. A continuación este valor se convierte a tipo *double*, y se le suma al valor de *d*. El resultado final es de tipo *double*.

Es posible forzar que una expresión pertenezca a un determinado tipo. Para ello podemos usar un operador de conversión de tipo llamado operador *cast*. Se usa de la siguiente forma:

```
(float) (x + y)
```

se está forzando a que el resultado de la expresión sea un *float*, independientemente del tipo de las variables *x* e *y*.

Ejemplos:

Expresión	Tipo de la expresión	Valor de la expresión
3 / 2	int	1
3.0 / 2	float	1.5
9/5 + 2.0	float	3.0
9.0/5 + 2	float	3.8
(float) (3/2)	float	1.0
(float) 3/2	float	1.5

3.9. Precedencia y asociatividad de operadores.-

Como ya se ha dicho en algún punto del capítulo, una expresión se evalúa siguiendo unas reglas de orden de prioridad (precedencia) entre operadores. Por ejemplo:

```
x + y * z /* Primero se hace la multiplicación y luego la suma */
```

En el caso de que dos operadores tengan la misma precedencia, se asociarán en un orden preestablecido, bien de izquierda a derecha, bien de derecha a izquierda. Por ejemplo, en el caso de los operadores de suma y resta, la asociatividad es de izquierda a derecha :

```
x + y - z /* Primero se hace la suma y luego la resta */
```

No obstante estas reglas pueden ser alteradas mediante el uso de paréntesis.

En la tabla 3.7 se muestran todos los operadores de C, con su precedencia de mayor a menor. También se muestra la asociatividad para aquellos operadores con la misma precedencia.

Operador	Asociatividad
() [] -> .	De izquierda a derecha
' - ++ -- - (tipo) * & sizeof	De derecha a izquierda
* / %	De izquierda a derecha
+ -	De izquierda a derecha
<< >>	De izquierda a derecha
<<= >>=	De izquierda a derecha
== !=	De izquierda a derecha
&	De izquierda a derecha
^	De izquierda a derecha
	De izquierda a derecha
&&	De izquierda a derecha
	De izquierda a derecha
?:	De derecha a izquierda
= += -= *= /= %= &= ^= \=	De derecha a izquierda
<<= >>=	
.	De izquierda a derecha

Tabla 3.7.- Precedencia y asociatividad de operadores en C.



## *Entrada y salida de datos por pantalla*

### 4.1. Introducción.-

Una de las cosas más importantes en un lenguaje de programación son las instrucciones que permiten una cierta interacción con el usuario. Tales operaciones suelen recibir el nombre de instrucciones de **Entrada/Salida**.

Suele ser este un tema que en otros libros se trata al final de los mismos, pero que aquí hemos decidido colocarlo al principio para poder hacer un correcto seguimiento de la mayoría de ejemplos expuestos en los sucesivos capítulos.

Sólo veremos unas pocas funciones, concretamente seis. Más adelante, en el capítulo 11, se ampliarán este grupo de funciones y todo lo relacionado con Entrada/Salida.

Los prototipos de las funciones que vamos a ver se encuentran en el fichero *stdio.h*, por lo tanto, lo primero que tendremos que hacer es incluirlo en cualquier programa que vaya a hacer uso de las funciones que vamos a ver.

## 4.2. Entrada y salida de caracteres.-

Para tratar los caracteres uno a uno, se puede hacer uso de las funciones `getchar()` y `putchar()`

### La función `getchar()`

Esta función devuelve un carácter leído del dispositivo de entrada estándar (normalmente el teclado)

```
char c;  
...  
c = getchar();
```

### La función `putchar()`

Sirve para visualizar un carácter en la salida estándar (normalmente el monitor).

```
char c;  
...  
putchar (c);
```

## 4.3. Entrada y salida de cadenas.-

Si lo que queremos es escribir o leer cadenas, las funciones que debemos usar son: `puts()` y `gets()`. Su uso es igual que `putchar` y `getchar()`, pero con cadenas.

Un ejemplo de cómo usarlas:

```
#include <stdio.h>  
  
main ()           /* Leer y escribir un línea de texto */  
{  
    char linea[80];  
  
    gets (linea);  
    puts (linea);  
}
```

## 4.4. Entrada y salida de datos formateada.-

Cuando queremos combinar datos numéricos con caracteres sueltos y con cadenas de caracteres, las funciones vistas hasta ahora se nos quedan cortas. Para poder escribir o leer datos con un determinado formato es necesario recurrir a otras funciones como son `printf()` y `scanf()`.

La función printf().-

En términos generales, la función `printf()` tiene la siguiente sintaxis:

```
printf (cadena de control, arg1, arg2, ..., argN)
```

donde *cadena de control* hace referencia a una cadena de caracteres que contiene información sobre el formato de salida y *arg1, arg2, ..., argN*, son argumentos que representan los datos de salida.

Lo explicaremos sobre un ejemplo. Consideremos la siguiente instrucción:

```
printf ("Me llamo %s y tengo %d años", minombre, miedad);
```

La cadena de control va entre comillas dobles. Está compuesta por caracteres normales y otros que van precedidos por el signo `%`. Los caracteres normales se imprimen tal cual, mientras que los otros tienen una interpretación. En el ejemplo que nos ocupa, cuando usamos `%s`, queremos decir que ahí se imprimirá una cadena, precisamente la que viene dada por *minombre*. Con `%d` queremos decir que ahí se imprimirá un entero, precisamente el que viene dado por *miedad*.

Como se puede ver, dependiendo de lo que queramos imprimir usaremos un carácter u otro siempre precedidos por el signo de `%`. Además de cadenas y enteros también podemos imprimir caracteres, punteros, números en coma flotante, etc. Para todos ellos existe un carácter que, precedido de `%`, hace que se pueda imprimir. En la tabla 4.1 se muestran los caracteres de conversión para la función `printf()`.

Carácter de conversión	Significado
c	El dato es visualizado como un carácter
d	El dato es visualizado como un entero decimal
e	El dato es visualizado como un valor en coma flotante con exponente
f	El dato es visualizado como un valor en coma flotante sin exponente
g	El dato es visualizado como un valor en coma flotante utilizando la conversión e ó f, la que sea más corta.
i	El dato es visualizado como un entero con signo
o	El dato es visualizado como un entero octal, sin el cero inicial
s	El dato es visualizado como una cadena de caracteres
u	El dato es visualizado como un entero decimal sin signo
x	El dato es visualizado como un entero hexadecimal sin el prefijo 0x

Tabla 4.1.- Caracteres de conversión para la función `printf()`.

Los caracteres especiales como retorno de carro, nueva línea, etc. también pueden introducirse en la cadena de formato. Así pues, en el siguiente ejemplo, primero se hará un salto a la siguiente línea y luego se imprimirá el mensaje:

```
printf ("\nEl Real Madrid es el campeón de Europa...");
```

Con saber esto de la función *printf()* de momento es suficiente. Ya ampliaremos más adelante algunos detalles que conviene saber.

### La función *scanf()*.-

Esta función es parecida a la anterior, pero para lectura de datos. Su sintaxis es:

*scanf (cadena de control, arg1, arg2, ..., argN)*

con la misma interpretación que antes.

Veamos algunos ejemplos, porque el tratamiento de los argumentos en esta función es distinto al tratamiento que se hace en *printf()*.

```
/* Lee una cadena y la almacena en la variable cadena */
scanf ("%s",cadena);
/* Lee un número y lo almacena en la variable numero */
scanf ("%d",&numero);
```

Como se puede ver hay una pequeña diferencia a la hora de leer una cadena o un número (o cualquier otra cosa). En C, el paso de parámetros a las funciones siempre es **por valor**, por lo que si queremos que la variable que pasamos como parámetro cambie su valor, debemos pasar la dirección de la misma. Esto se consigue con el operador unario **&**.

No se preocupe el lector si no entiende esto último muy bien. En los capítulos dedicados a las funciones y a los punteros, se encontrarán los conocimientos necesarios para entenderlo. De momento basta con saber que para leer cadenas no hace falta pasar su dirección, pero para cualquier otro tipo sí que hace falta y para ello usaremos el operador **&**. En el siguiente ejemplo se leen sucesivamente una cadena, un entero y un número en coma flotante.

```
scanf ("%s %d %f",cadena, &entero, &flotante);
```

y una posible entrada de datos podría ser:

```
hola
6
2.5
```

Los caracteres de conversión son similares a los vistos en *printf()*.

Con lo visto hasta ahora de Entrada/Salida de datos es suficiente para poder seguir los ejemplos que se muestran en capítulos posteriores, que era el objetivo fijado en principio.



## *Control del flujo*

### 5.1. Introducción.-

El flujo de ejecución es el orden en el que se van ejecutando las instrucciones de un programa. Como ya sabemos, en un programa secuencial, las instrucciones se van ejecutando una detrás de la otra. Nosotros podemos cambiar este orden de ejecución mediante las ***estructuras de control del flujo***. Estas estructuras nos permiten modificar el flujo de ejecución secuencial en dos sentidos :

- Por un lado nos permiten decidir sobre la ejecución de un conjunto de instrucciones en base a la evaluación de una condición. Esto lo podemos conseguir a través de las ***estructuras condicionales***.
- Por otro lado, nos permiten ejecutar repetidas veces un mismo conjunto de instrucciones dependiendo de la evaluación de una condición. Esto lo conseguimos gracias a las ***estructuras iterativas***.

Como podemos comprobar, en ambos casos entra en juego la evaluación de condiciones. Veamos pues cómo podemos expresar estas condiciones.

## 5.2. Condiciones.-

Una *condición* en C no es más que una expresión de cualquier tipo encerrada entre paréntesis. Una condición puede estar formada por operadores relacionales, lógicos y aritméticos.

Cualquier expresión en C con un valor **igual a 0** será evaluada como FALSA, en otro caso, es decir, cualquier expresión con valor **distinto de 0**, es evaluada como CIERTA. Pueden ser ejemplos de condiciones los siguientes :

<code>(x&lt;y)</code>	cierta cuando x es menor que y.
<code>(x&lt;y &amp;&amp; x&gt;z)</code>	cierta cuando x es menor que y, pero mayor que z
<code>(x!=y)</code>	cierta cuando y es distinto de 0
<code>(x==y)</code>	cierta cuando x e y son iguales
<code>(x-y)</code>	cierta cuando x e y son distintas

Veamos ahora cómo podemos agrupar un conjunto de instrucciones de forma que éstas puedan ser ejecutadas todas dependiendo de una condición tal y como hemos apuntado en la introducción.

## 5.3. Instrucciones simples y compuestas.-

Una instrucción simple en C es toda expresión seguida de un punto y coma ';'.  
Por ejemplo :

```
x = y;
x--;
```

Una instrucción compuesta resulta de la unión de varias instrucciones simples encerrándolas entre llaves '{ }'. Por ejemplo :

```
{
    x = y;
    x--;
}
```

Como particularidad hemos de apuntar que al inicio de una instrucción compuesta, es decir, después de los paréntesis, se permiten declaraciones de variables cuyo alcance será el de la instrucción compuesta. De este modo, las variables así declaradas desaparecerán al ejecutarse instrucciones que están fuera de los paréntesis.

Las instrucciones compuestas nos permiten que, dada la evaluación de una condición, podamos ejecutar **un grupo** de instrucciones u otro, gracias a las estructuras condicionales, una o varias veces, gracias a las estructuras iterativas, en lugar de que lo que se ejecute sea **sólo una** instrucción.

Veremos a continuación las diferentes estructuras condicionales que nos proporciona el lenguaje C.

## 5.4. Estructuras condicionales.-

### 5.4.1. Estructura if...else

Esta estructura permite elegir entre la ejecución de una instrucción u otra, sean éstas simples o compuestas, en base a la evaluación de una condición. Su sintaxis es la siguiente :

```
if (condición)
    instrucción 1;
else
    instrucción 2;
```

El funcionamiento sería el siguiente: se evalúa *condición*, si el resultado es CIERTO, se ejecuta *instrucción 1*, si el resultado es falso, se ejecuta *instrucción 2*. Una vez que ha terminado la ejecución de la instrucción 1 ó 2, el flujo del programa sigue normalmente.

Veamos un ejemplo :

```
if (x == y)
    printf ("las variables son iguales\n");
else {
    x--;
    printf ("las variables son distintas y he decrementado x\n");
}
```

En este ejemplo comprobamos si *x* e *y* son iguales, en caso afirmativo se imprime un mensaje que indica que las variables son iguales; en caso contrario, se decrementa *x* y se muestra el correspondiente mensaje.

Podemos omitir el *else* en la estructura condicional y entonces la sintaxis quedaría de la siguiente forma :

```
if (condición)
    instrucción;
```

Por ejemplo :

```
if (x == y)
    printf ("las variables son iguales\n");
```

En este ejemplo sólo comprobamos si *x* e *y* son iguales, y en caso de que así sea, se imprime el mensaje.

Podemos anidar varias estructuras *if...else*, nada nos impide que *instrucción 1* ó *instrucción 2* sean una estructura *if* o sean instrucciones compuestas que contengan estructuras *if*. Por ejemplo:

```

if (x > 0)
{
    if (x<5)
        printf ("x es mayor que cero pero menor que cinco\n");
    else
    {
        x--;
        printf ("x es mayor que cinco y la he decrementado\n");
    }
    printf ("He ejecutado la primera parte del if.\n");
    printf ("La condicion era cierta.\n");
}
else
    if (x > -5)
        printf ("x es menor que cero pero mayor que -5\n");

```

Este ejemplo comprueba si la variable *x* es mayor que cero; en caso afirmativo comprueba si la variable es menor que cinco, en cuyo caso lo comunica con un mensaje. Si *x* por el contrario es mayor que cinco, la decrementa y lo comunica. En caso de que *x* fuese menor que cero, comprueba si es mayor que menos cinco y si es así lo comunica.

La anidación de estructuras *if...else* puede presentar ambigüedades cuando, una vez que se han evaluado las condiciones, lo que se ejecutan son instrucciones simples (es decir, no empleamos las llaves). Veamos el siguiente ejemplo:

```

if (x > 0)
    if (x>5)
        printf ("x es mayor que cero pero menor que cinco\n");
else
    printf ("¿cómo es la variable?\n");

```

¿A qué *if* corresponde el *else*, al primero o al segundo?, tal y como están igual podría corresponder a uno que a otro, pero en realidad, pertenece al más interior. Así, en este caso, la parte del *else* se ejecutará si la variable es mayor que cero pero menor que cinco.

En C, los *else* se asocian a los *if* de dentro hacia fuera. Para deshacer ambigüedades se aconseja el uso de las llaves. Si en el ejemplo anterior quisiéramos que el *else* correspondiera al *if* (*x>0*) lo haríamos de la siguiente forma:

```

if (x > 0)
{
    if (x<5)
        printf ("x es mayor que cero pero menor que cinco\n");
}
else
    printf ("¿cómo es la variable?\n");

```

### 5.4.2. El operador condicional

El *operador condicional* es una estructura similar a la *if...else*, en un formato más comprimido, y que en lugar de permitirnos elegir entre la ejecución de una instrucción u

otra nos permite elegir entre la evaluación de una expresión u otra. Su sintaxis es la siguiente :

*(condición) ? expresión1 : expresión2;*

Para ejecutarse el operador condicional, primero se evalúa *condición*, si es cierta se evalúa *expresión1*, si por el contrario es falsa, se evalúa *expresión2*.

Este operador es de especial utilidad cuando se quiere hacer una asignación condicional. Con una estructura *if....else*, esta operación se haría de la siguiente forma :

```
if (x>0)
    y = x + 3;
else
    y = (-1 * x) + 3
```

Con estas instrucciones asignamos a la variable *y* el valor absoluto de la variable *x* aumentado en tres unidades. Podríamos hacer lo mismo con el operador condicional de una forma más compacta :

```
y = (x>0) ? x+3 : (-1*x) + 3;
```

Al igual que las estructuras *if....else* se pueden anidar, también podemos anidar operadores condicionales :

```
— y = (x>0) ? x+3 : ((x>-5) ? (-1*x)+1 : (-1*x) +10);
```

Con este operador condicional analizamos el valor de la variable *x*, si *x* es mayor que cero se le asigna su valor a la variable *y* aumentado en tres unidades; si *x* es menor o igual que cero, pero mayor que menos cinco, se asigna a la variable *y* el valor absoluto de *x* aumentado en una unidad; si *x* es menor o igual que menos cinco se asigna a la variable *y* el valor absoluto de *x* aumentado en diez unidades.

### 5.4.3. Estructura switch

La estructura *switch* nos permite escoger entre distintos fragmentos de código para ejecutarlos dependiendo del valor de una expresión. Esto mismo se puede conseguir con el anidamiento de estructuras *if....else*, pero el lenguaje C, nos proporciona este método que es más sencillo y compacto.

La sintaxis de la estructura *switch* es la siguiente:

```
switch (expresión)
{
    case valor1:                /* etiqueta case */
        instrucciones 1;
    case valor2:
        instrucciones 2;
    .....
    case valor n:
        instrucciones n;
    default :                  /* etiqueta default */
        instrucciones;
}
```

En esta estructura *valor1*, *valor2*, ....., *valor n* han de tomar valores constantes, es decir, no pueden ser en ningún caso expresiones que incluyan variables. *instrucciones 1*, *instrucciones 2*, ....., *instrucciones n*, son secuencias de instrucciones que a su vez pueden constar de una o más instrucciones.

La estructura *switch* se ejecuta de la siguiente forma: en primer lugar se evalúa la expresión, una vez que se ha evaluado, el control se transfiere a la *etiqueta case* que coincide con el valor de la expresión. Si no hay ninguna etiqueta *case* que coincida con el valor de la expresión, el control se transfiere a la *etiqueta default*. Una vez que el control ha sido transferido a una etiqueta se ejecuta la instrucción asociada a esa etiqueta y el programa continúa secuencialmente, lo que quiere decir que si por debajo de esa instrucción hay más etiquetas e instrucciones también se ejecutarán. Por ejemplo, en la estructura *switch* :

```
switch x
{
    case 1 :
        printf ('el valor de la variable es 1\n');
    case 2 :
        printf ('el valor de la variable es 2\n');
    default :
        printf ('el valor de la variable no es ni 1 ni 2\n');
}
```

Si el valor de *x* fuera 1 el control se transferiría a la etiqueta *case 1* y se imprimiría el mensaje '*el valor de la variable es 1*', pero luego se seguiría ejecutando secuencialmente el programa y se imprimiría los mensajes '*el valor de la variable es 2*' y '*el valor de la variable no es ni 1 ni 2*'. Para evitar este comportamiento podemos utilizar la **sentencia break**. Esta sentencia hace que cuando se llega a ella el control pase a la siguiente sentencia que hay tras la llave que cierra la estructura *switch*.

Por ejemplo:

```
switch x
{
    case 1 :
        printf ('el valor de la variable es 1\n');
        break;
    case 2 :
        printf ('el valor de la variable es 2\n');
        break;
    default :
        printf ('el valor de la variable no es ni 1 ni 2\n');
}
```

En este caso cuando la variable *x* vale 1 se visualiza el mensaje '*el valor de la variable es 1*', cuando *x* vale 2 se visualiza el mensaje '*el valor de la variable es 2*' y cuando *x* vale otra cosa se visualiza el mensaje '*el valor de la variable no es ni 1 ni 2*'.

La sentencia *break* se utiliza además para otros propósitos como ya veremos a lo largo del capítulo. Esta sentencia se incluye dentro de las sentencia de transferencia incondicional de flujo, es decir, transfiere el control del programa de un punto a otro de forma incondicional. Por ello, no se aconseja su uso más que en casos en los que no tengamos más opción. Debe evitarse aplicando técnicas de la programación estructurada.

## 5.5. Estructuras iterativas.-

### 5.5.1. Estructura while

La estructura *while* nos permite ejecutar una instrucción, ya sea ésta simple o compuesta, repetidas veces *mientras se cumpla una determinada condición*. Su sintaxis es la siguiente:

```
while (condición)
    instrucción;
```

Esta estructura funcionaría de la siguiente forma: en primer lugar se evalúa *condición*, si es falsa, se pasa a la siguiente instrucción después de la estructura *while*; si es cierta se ejecuta *instrucción* y volvemos al principio, es decir, se vuelve a evaluar la *condición*, si es falsa ...

Por ejemplo, podemos calcular el factorial de un cierto número de la siguiente manera :

```

if (x >= 0)
{
    i = 1;
    factorial = 1;
    while (x >= i)
    {
        factorial *= i;
        i ++;
    }
}
else
    printf ("No existe el factorial de un número negativo.\n");

```

Esta estructura también puede ser interrumpida mediante la sentencia *break*. Cuando eso ocurre, el control del programa, pasa a la siguiente instrucción después de la estructura. No se aconseja su utilización pues nos introduce en un método de programación no estructurada y siempre puede ser sustituida ampliando la condición de la estructura. El siguiente ejemplo :

```

while (a)
{
    ..... /* Instrucciones 1 */
    if (b) break;
    ..... /* Instrucciones 2 */
}

```

Podría ser sustituido por el que a continuación se detalla, sin utilizar la sentencia *break* y consiguiendo el mismo resultado:

```

while (a && !(b))
{
    ..... /* Instrucciones 1 */
    if (!b)
    {
        ..... /* Instrucciones 2 */
    }
}

```

### 5.5.2. Estructura do....while

Esta estructura es similar a la estructura *while* con la diferencia de que la evaluación de la condición se realiza *después* de haber ejecutado *instrucción*, por lo tanto, *instrucción* se ejecuta siempre al menos una vez.

Su sintaxis es la siguiente :

```

do
    instrucción;
while (condición);

```



Su funcionamiento sería el siguiente : cuando la ejecución llega a *do*, se ejecuta *instrucción*, a continuación se evalúa *condición*, si es falsa sigue con la siguiente instrucción del programa; sino, vuelve al principio, es decir, se vuelve a ejecutar *instrucción*, se evalúa *condición*, etc...

El siguiente ejemplo de estructura *do....while* calcula el factorial de un número:

```
if (x >= 0)
{
    i = factorial = 1;
    do
    {
        factorial *= i;
        i++;
    }
    while (x <= 1);
}
else
    printf ("no existe el factorial de un número negativo.\n");
```

Hemos de notar que una estructura *while* siempre se puede convertir en una estructura *do....while* y viceversa. Por lo tanto, utilizar una u otra en cada caso dependiendo de lo que nos sea más cómodo.

Al igual que *switch* y *while*, la estructura *do....while* también puede ser interrumpida por la sentencia *break* aunque no se recomienda su utilización.

### 5.5.3. Estructura for

Esta estructura es también similar a *while* aunque es más apropiada que ésta cuando *condición* involucra a un contador que se actualiza dentro del bucle. Su sintaxis es la siguiente:

```
for (inicialización ; condición ; actualización)
    instrucción;
```

Tanto *inicialización* como *actualización* pueden ser expresiones de cualquier tipo, aún así, *inicialización* suele ser una sentencia de asignación, *actualización* suele ser una sentencia que modifique el valor de la variable inicializada y *condición* una condición que involucre a dicha variable.

El funcionamiento de la estructura es el siguiente: se evalúa *inicialización*, a continuación se evalúa *condición*, si es falsa, se acaba la ejecución de la estructura y se continua con la siguiente instrucción del programa, si es cierta se ejecuta *instrucción*, a continuación se evalúa *actualización* y se vuelve a evaluar *condición*, a partir de aquí se vuelve a repetir el proceso.

Para calcular el factorial con una estructura *for* :

```
if (x >= 0)
{
    factorial = 1;
    for (i = 1 ; i <= x ; i++)
        factorial *= i;
}
else
    printf ("No existe el factorial de un número negativo.\n");
```

Toda estructura *for*, puede convertirse a una estructura *while* o *do....while* y viceversa.

Cualquiera de las tres expresiones de la cabecera de una estructura *for*, léase *inicialización*, *condición* y *actualización*, pueden ser omitidas. En caso de que la omitida fuese la condición ésta se evaluaría siempre como cierta. Ejemplos :

```
for ( ; ; )
    printf ("estoy escribiendo esto eternamente.\n");

for (i = 0 , ; i++)
    printf ("voy por el nº ",i," en mi camino hacia infinito.\n");
```

Además, en las expresiones inicialización y condición se puede utilizar carácter coma ',' con el objeto de permitir la inclusión de más de una sentencia :

```
for (i = 0 , j = 1000 ; i != j ; i++ , j--)
    printf ("las variables i y j todavía no son iguales.\n");
```

La estructura *for* también puede ser interrumpida por la sentencia *break*.

## 5.6. Sentencia goto.-

La sentencia *goto* nos permite transferir el control del programa de un punto a otro de forma incondicional. Su sintaxis es la siguiente:

```
goto etiqueta;
..... /* Instrucciones 1 */
etiqueta:
..... /* Instrucciones 2 */
```

Cuando el programa llega a la sentencia *goto etiqueta*, salta a *etiqueta* sin ejecutar *instrucciones 1* y comienza a ejecutar la primera instrucción de *instrucciones 2*, es decir, la siguiente instrucción después de *etiqueta*.

*Etiqueta* ha de ser un nombre único dentro del programa, puede encontrarse tanto antes como después de la sentencia *goto* y siempre ha de haber alguna instrucción después de ella, aunque ésta sea la instrucción vacía (;).

No se recomienda el uso de esta sentencia pues nos lleva a una programación no estructurada, además, siempre puede ser sustituida por las sentencias de control de flujo que hemos estudiado durante el presente capítulo.

Por ejemplo, el siguiente programa:

```
main ( )
{
    ..... /* Instrucciones 1 */
    if (x >= 0)
    {
        z = x;
        goto factorial;
    }
    else
        goto error;
    ..... /* Instrucciones 2 */
    if (y >= 0)
    {
        z = y;
        goto factorial;
    }
    else
        goto error;
    factorial :
        ..... /* Instrucciones de cálculo del factorial */
        goto final;
    error :
        ..... /* Instrucciones de tratamiento de error */
    final:
        return;
}
```

Podría sustituirse por el siguiente:

```
main_1 ( )
{
    ..... /* Instrucciones 1 */
    if (x >= 0)
        z = rutina_factorial (x);
    else
        rutina_error ();
    ..... /* Instrucciones 2 */
    if (y >= 0)
        z = rutina_factorial (y);
    else
        rutina_error ();
    return;
}
```

## 5.7. Sentencia continue.-

La sentencia *continue* se utiliza dentro de las estructuras iterativas para forzar que se salten las instrucciones que quedan entre ella y la próxima evaluación de la condición, y se pase a evaluar directamente esta condición.

Su sintaxis es la siguiente :

```
while (condición 1)  
  {  
    ..... /* Instrucciones 1 */  
    if (condición 2) continue;  
    ..... /* Instrucciones 2 */  
  }
```

Cuando se evalúa *condición 2*, si es cierta se vuelve al principio del bucle y se vuelve a evaluar *condición 1* sin ejecutar *instrucciones 2*.

Al igual que la sentencia *break*, la sentencia *continue* provoca un salto incondicional en el flujo de control. Por ello, y porque siempre puede ser sustituida, se aconseja no utilizarla.

### 6.1. Introducción.-

En el presente capítulo estudiaremos las herramientas que nos proporciona el lenguaje C para programar modularmente.

Como ya conocerá el lector, muchos lenguajes de programación permiten la utilización de dos tipos de módulos a los programadores: las *funciones*, que devuelven un valor para ser utilizado dentro de una expresión (por ejemplo la asignación del valor devuelto a una variable) y los *procedimientos*, que agrupan un conjunto de instrucciones bajo el nombre de una pseudo instrucción; normalmente, este conjunto de instrucciones, será utilizado en diferentes lugares del programa ejecutándose sobre distintos datos. Este es el caso, por ejemplo, del Pascal.

El lenguaje C sin embargo nombra a los dos tipos genéricamente funciones. El motivo es que, originariamente, en C todos los módulos tenían que devolver algún valor. De ellos, unos nos interesarían y serían almacenados en variables, como por ejemplo el valor devuelto por la función que calcula el módulo; y otros no nos interesarían, como por ejemplo el valor devuelto por la función que visualiza una cadena en el monitor.

Además, el lenguaje C, distingue entre la *declaración* de una función y su *definición*.

La declaración de una función es la especificación de su nombre, del tipo de sus parámetros y del tipo del valor que devuelve, no incluye el cuerpo de la función y se utiliza para indicar al compilador que más tarde se definirá una función con el nombre y el tipo de parámetros especificados.

La definición de una función incluye el nombre de ésta, el nombre y tipo de los parámetros, el tipo del valor devuelto y la secuencia de instrucciones que forman el cuerpo de la función. Estudiaremos a continuación la sintaxis de cada una de estas acciones.

## 6.2. Definición de funciones.-

Para la definición de una función en C podemos utilizar dos sintaxis diferentes, por una parte, la sintaxis definida por Kernighan y Ritchie, y por otra, la definida por el estándar ANSI, incluida con posterioridad para facilitar la comprobación de errores en el paso de parámetros. La sintaxis K&R es la siguiente:

```

tipo_resultado nombre_función (parámetro_1 , parámetro_2 , ..... )
    tipo_parámetro_1 parámetro_1;
    tipo_parámetro_2 parámetro_2;
    .....
{
    cuerpo de la función
}
```

La sintaxis establecida por el estándar ANSI fue la siguiente:

```

tipo_resultado nombre_función (tipo_parámetro_1 parámetro_1 ,
                                tipo_parámetro_2 parámetro_2 , ..... )
{
    cuerpo de la función
}
```

En nuestros programas podemos emplear cualquiera de las dos sintaxis especificadas. No obstante es aconsejable utilizar la sintaxis del ANSI C. De cualquier forma, sea cual sea la sintaxis empleada, en la definición de una función entran en juego cuatro elementos importantes: el **tipo** del resultado, el **nombre** de la función, los **parámetros y su tipo** y el **cuerpo** de la función. Estudiaremos a continuación cada uno de estos elementos.

## Tipo del resultado de una función .-

El tipo de una función se especifica delante del nombre de la función y representa el tipo de valor que la función devolverá:

```
int lee_caracter ()
{
    ...
}
```

Si el tipo de una función no se especifica, se supone que ésta es de tipo *int*.

```
lee_caracter ()
{
    ...
}
```

Cuando usemos funciones que no necesitan devolver un valor es aconsejable el uso del tipo *void*:

```
void error (void)
{
    printf ("Has cometido un error\n");
}
```

Las funciones tipo *void* sólo podremos usarlas en expresiones del tipo:

```
funcion ( );
```

La utilización de una función de este tipo en una asignación daría lugar a un error.

## El nombre de una función.-

El nombre de una función está sujeto a las normas que vimos en el capítulo 2 para los identificadores.

## Parámetros formales.-

Nos quedaremos con la forma de definir las funciones del tipo ANSI C:

```
int potencia (int base, int exponente)
{
    ... /* Función que devolverá un tipo entero */
}
```

• En el momento de la llamada a la función, los parámetros reales se copian en los parámetros formales. Se trata de una **llamada por valor**. El cambio que se haga de los parámetros dentro de la función no afecta a los parámetros reales. Existe una manera de

conseguir pasar parámetros **por referencia** con el uso de punteros, tema que veremos en el capítulo 8.

```
main ( )
{
    ...
    Resultado = potencia (x,y);
    ...
}
```

En principio, todas las variables declaradas en una función son locales, es decir, no son conocidas fuera de ella. Los parámetros son también locales a la función, motivo por el cual, aunque se modifique un parámetro dentro de la función, la copia original queda inalterada. En el siguiente ejemplo se mostrará el valor 3:

```
main ()
{
    int x = 3;

    suma_2 (x);
    printf ("%d\n",x);
}

void suma_2 (int x)
{
    x += 2;
}
```

Si queremos que una función no tome argumentos, podemos especificarlo de 2 formas:

- utilizando paréntesis vacíos: ( )
- especificando *void* entre los paréntesis: (void)

Ejemplos:

```
void sin_param ()
{ ... }

void sin_param_2 (void)
{ ... }
```

### 6.3. Valor devuelto por una función.-

Existe una instrucción especial que permite a una función devolver un valor:

*return expresion;*

*return* devuelve el control a la función llamadora junto con el valor de la *expresión*.

Varias cosas a tener en cuenta a la hora de usar *return*:



1. **return** no es una función sino una palabra clave del C, por lo tanto no necesita paréntesis como las funciones, aunque también es correcto: `return (expresion)`.
2. El tipo de la expresión utilizada con *return* debe ser el mismo que el de la función.
3. No utilizar *return* si no se va a devolver ningún valor. Al llegar al final de la función el control vuelve siempre a la función llamadora.
4. Declarar las funciones como pertenecientes al tipo *void* en caso de que no se pretenda devolver ningún valor; de esta forma el compilador puede detectar un mal uso de *return* o de la función.

Veamos un ejemplo:

```
#include <stdio.h>

int maximo (int ma, int mb);      /* explicado en la      */
long potencia (int pa, int pb);  /* siguiente sección */

void main (void)
{
    int a=2, b=3, c=4, d=5;

    printf ("\nEl máximo entre %d y %d es %d ", a,b,maximo(a,b));
    printf ("\n%d elevado a %d es %d.\n", c, d, potencia (c,d));
}

int maximo (int ma, int mb)
{
    return (ma >= mb) ? ma : mb;
}

long potencia (int pa, int pb)
{
    int i;
    long pot = 1;

    for (i = 1; i <= pb; i++)
        pot *= pa;
    return pot;
}
```

La salida del programa sería:

```
El máximo de 2 y 3 es 3.
4 elevado a 5 es 1024.
```

## 6.4. Declaración de funciones (prototipos).-

El C permite declarar funciones antes de su definición. En el ejemplo de la sección anterior se puede ver la declaración de las funciones *potencia* y *maximo*. La declaración de una función tiene 2 propósitos principales:

- Evitar conversiones erróneas de valores.

- Evitar errores en número o tipo de argumentos.

## 6.5. La función *main* ( ).-

Como ya se ha comentado en algún momento, todo programa en C necesita una función *main()*. Será por esta función por donde empezará a ejecutarse el programa. Como toda función, puede tener un tipo y una lista de parámetros formales asociados.

### Argumentos de *main* ( ).-

Algunas veces es útil pasar información al programa cuando se va a ejecutar. Esto se hace colocando los argumentos a continuación del nombre del programa en la línea de órdenes del sistema operativo:

*NombrePrograma [argumento1, ....]*

Ejemplo:        *Listar Fichero1, Fichero2*

Para este caso, la función *main()* se declara con dos parámetros especiales, *argc* y *argv*, que se utilizan para recibir argumentos desde la línea de órdenes.

El parámetro *argc* contiene el número de argumentos de la línea de órdenes y es un entero. Siempre vale como mínimo 1, pues el nombre del programa cuenta como el primer argumento.

El parámetro *argv* es un array donde cada elemento es una cadena de caracteres. Cada elemento será uno de los argumentos pasado en la línea de órdenes del S.O.

Los nombres *argc* y *argv* son una convención, pueden sustituirse por cualquier nombre válido de un identificador en C.

Vamos a ver un ejemplo del uso de estos argumentos. No obstante, en el capítulo 8 dedicado a los arrays volveremos sobre el particular.

El siguiente programa, acepta un nombre desde la línea de comandos, justo a continuación del nombre del programa, e imprime un mensaje de saludo :

```
#include <stdio.h>

void main (int argc, char *argv[])
{
    if (argc != 2)
        printf ("El número de argumentos es incorrecto\n");
    else
        printf ("Hola %s.\n",argv[1]);
}
```

## 6.6. Funciones de la librería estándar de C.-

En algunos ejemplos que llevamos visto aparecen líneas de código comenzadas por *#include*, que es una directiva que permite acceder a cualquier función estándar almacenada en la librería de C.

Por ejemplo, *#include <stdio.h>* permite acceder a cualquier función que esté declarada en el fichero *stdio.h* y que corresponde con la familia de funciones de entrada/salida estándar.

Nosotros podemos usar cualquier función que el compilador de C traiga definida, siempre y cuando hagamos un *#include* del fichero donde se encuentra declarada, normalmente con extensión *.h*

Para más información acerca de las librerías de funciones, consultar el apéndice B, dedicado por completo a las librerías del C. De igual forma, para más detalles sobre la directiva *#include*, consultar el capítulo 13.

## 6.7. Funciones con número variable de argumentos.-

El C nos permite declarar una función con un número indeterminado de argumentos, de forma que en cada llamada a la función, puedan pasarse diferente número de argumentos.

Ya hemos visto alguna de estas funciones. Los dos casos más típicos son las funciones *printf()* y *scanf()*. Si recordamos, por ejemplo para el caso de *printf()*, podemos escribir el valor de un número indeterminado de variables, que no tiene porqué ser en todas las llamadas el mismo.

La sintaxis de la declaración de una función con número variables de argumentos es la siguiente :

*tipo\_devuelto nombre ({argumentos\_fijos}, ...);*

donde *argumentos\_fijos* puede omitirse, pero en caso de ir debe indicar el tipo de los argumentos dentro de los parámetros.

Ejemplo :

```
void prueba1 (char *mensaje, ...);
```

Es necesario incluir el fichero `<stdarg.h>`, donde se encuentran las definiciones de las funciones auxiliares que nos permiten manejar la lista de argumentos.

Los argumentos de la función se almacenan en una lista del tipo `va_list`, definido en el fichero anteriormente reseñado. En el cuerpo de nuestra función tenemos que declarar una lista de este tipo. Por ejemplo :

```
void prueba1 (char *mensaje, ...)
{
    va_list lista;
    ...
}
```

Nuestra función debe comenzar con una llamada a `va_star()`, que inicializa el contenido de la lista de argumentos. Su sintaxis es :

*void va\_star (lista\_argumentos, nombre\_ultimo\_argumento\_fijo);*

Luego, para obtener cada uno de los argumentos, tenemos que llamar a la función `va_arg()`, que retorna el primer argumento interpretado con el *tipo* dado. En cada llamada se devuelve uno. Su sintaxis es :

*tipo va\_arg (lista\_argumentos, tipo);*

Finalmente, se termina llamando a la función `va_end()`, cuya sintaxis es:

*void va\_end (lista\_argumentos);*

A continuación, para finalizar el capítulo, vamos a ver dos ejemplos de funciones con número variable de argumentos. El primero es más sencillo de entender, mientras que el segundo hace referencia a cosas que estudiaremos en capítulos posteriores. Se recomienda que después de haber visto los capítulos dedicados a punteros y arrays se vuelva sobre este ejemplo, ya que en ese momento se podrán entender todos los conceptos que en él se muestran.

## Ejemplo 1.

```

/*
En este ejemplo se implementa la función suma, que devuelve la suma de un
numero indeterminado de enteros positivos. Nótese que dada la implementación
que se hace, es posible que el primer sumando sea negativo.
*/

#include <stdarg.h>
#include <stdio.h>
#include <string.h>

/* Prototipo de la función */
int suma (int i,...);

main()
{
    printf ("\n\n suma ->   %d ", suma (10,5,15,3,78,9,5,3,7,9) );
    printf ("\n\n suma ->   %d ", suma (2) );
    printf ("\n\n suma ->   %d ", suma (-10,5,12) );
}

int suma (int i,...)
{
    int suma=1, x;

    va_list lista;
    va_start (lista, i);
    x = va_arg (lista, int);
    while (x > 0)
    {
        suma = suma + x;
        x = va_arg (lista, int);
    }
    va_end (lista);
    return (suma);
}

```

Para las distintas llamadas, el resultado en pantalla sería :

```

suma -> 144
suma -> 2
suma -> 7

```

## Ejemplo 2.

```
/*
La función strcat_var concatena todas las cadenas pasadas como argumentos,
dejando el resultado en la variable destino. Para que funcione
correctamente, el último argumento debe ser NULL.
PUEDE QUE EL LECTOR NO ENTIENDA ALGUNAS COSAS DE ESTE EJEMPLO.
VUELVA A MIRARLO DESPUES DE ESTUDIAR LOS CAPITULOS 8 Y 9.
*/
```

```
#include <stdarg.h> /* va_start(), va_arg(), va_end() */
#include <stdio.h> /* NULL */
#include <string.h> /* strcat() */
```

```
/* Prototipo de la función */
void strcat_var (char *destino,...);
```

```
main ()
{
    char resultado[100] = ""; /* la inicializamos vacia */

    strcat_var (resultado, "hola","queridos","chavales.", NULL);
    printf ("\n\n%s", resultado);
    getchar();
}
```

```
void strcat_var (char *destino,...)
{
    char *t;
    va_list lista;

    va_start (lista,destino);
    t = va_arg (lista, char*);
    while (t != NULL)
    {
        strcat (destino, t);
        t = va_arg (lista, char*);
    }
    va_end (lista);
}
```

El resultado en pantalla de la ejecución del programa sería :

hola queridos chavales.

## *Ámbito de las variables*

### **7.1. Introducción.-**

Todas las variables de un programa tienen un ámbito de aplicación, entendido como la porción de código para la que es válida su declaración. Fuera de este ámbito, una variable no puede ser usada por ninguna función, el compilador no la reconocería. El ámbito de aplicación también es conocido como **tiempo de vida**.

Básicamente las variables, y en general todos los objetos que podemos definir en C, van a tener un ámbito que dependerá en principio del lugar donde sean declarados, aunque esto puede cambiarse siempre haciendo uso de modificadores.

## 7.2. Tiempo de vida de los objetos.-

Antes de nada hay que decir que se entiende por objeto cualquier dato o código que ocupa memoria direccionable. Los objetos incluyen variables, funciones, punteros, arrays, cadenas, estructuras y uniones (estas 2 últimas ya las veremos más adelante en el capítulo 10).

El tiempo de vida de un objeto puede ser global o local :

- Tiempo de vida **global** significa que una vez que el objeto es definido, permanece hasta que el programa termina.
- Tiempo de vida **local** significa que se asigna memoria al objeto cada vez que el control de programa pasa a través del bloque de código en el que el objeto está definido. Después de que el control sea devuelto por el bloque, la memoria ocupada por el objeto se libera y el valor del objeto queda indefinido.

El tipo de vida de un objeto está determinado por **cómo y dónde** está declarado :

- El tiempo de vida de todas las funciones es global. En C no se permite el anidamiento de funciones.
- El tiempo de vida de las variables declaradas en un nivel externo (fuera de un bloque, por ejemplo, antes de la función *main*) es siempre global.
- El tiempo de vida de las variables declaradas en un nivel interno (dentro de un bloque) es local, a menos que la declaración contenga un **tipo de almacenamiento** especificado que supedite las reglas habituales.

## 7.3. Especificaciones de tipos de almacenamiento.-

El tipo de almacenamiento viene dado por una de las cuatro palabras clave (**auto**, **register**, **static** y **extern**) que determinan el tiempo de vida de un objeto.

### Auto.-

Este tipo de almacenamiento tiene lugar únicamente en el interior de un bloque y declara la variable como local. Puesto que **auto** se permite sólo en un nivel interno y las variables definidas en ese nivel son automáticas por defecto, la palabra clave *auto* es redundante y rara vez se emplea. Ejemplo:

```
void funcion()
{
    auto int i;    /* Variable local */
    char c; /* Variable local */
    ....
}
```



## Register.-

Cuando tenemos variables locales que se usan mucho dentro de un bloque es posible reducir el tiempo de ejecución del programa declarando estas variables como de tipo **register**. Estas variables se almacenan en registros de la CPU en vez de hacerlo en memoria, por lo que el tiempo de acceso a ellas es menor. Si no existiese ningún registro libre en la CPU, se cargarían en memoria, como si fuesen de tipo *auto*. Las variables *register* deben ser enteros.

Ejemplo :

```
register int i;  
for (i = 0; i < 1000 ; ++i)  
    ...
```

Es típico usar estas variables para almacenar los contadores de bucles.

## Static.-

A veces, aunque una variable únicamente es necesaria dentro de una sola función, es preciso que su valor se conserve de una llamada a otra. Este tipo de variables se denominan variables static.

Para convertir una variable automática en *static*, basta con poner delante de la declaración de la variable la palabra reservada *static*. Por ejemplo:

```
static int i;
```

Un problema que se presenta con las variables *static* es el de su inicialización. Si hiciéramos:

```
static int i;  
i = 1;
```

no tendría ninguna utilidad que la variable *i* fuese estática, puesto que cada vez que se llamara a la función se volvería a realizar la asignación. La solución es realizar la inicialización de las variables *static* en el momento de su declaración:

```
static int i=1;
```

De esta forma, el compilador reconoce que la asignación sólo debe realizarse la primera vez que se invoca a la función.

## Extern.-

Por cuestiones de modularidad y diseño, a veces es conveniente distribuir el código fuente en varios ficheros. Dado que el ámbito de una variable se reduce al fichero fuente donde ha sido declarada, en el resto de los ficheros tendremos que declarar explícitamente que deseamos usar una variable definida en un lugar distinto del fuente actual (una variable externa). Para ello, el lenguaje C nos permite usar la palabra reservada **extern** al comienzo de la declaración.

Una declaración de la forma :

```
extern int i;
```

indica al compilador que la variable *i* ha sido declarada en otro fuente, y que no es necesario volver a reservar espacio físico. Si una variable debe ser compartida por varios ficheros fuente, todos ellos **excepto uno** deben contener la declaración *extern*, y sólo uno de ellos la declaración sin *extern*.

Ejemplo:

### Fuente 1:

```
char c;    /* Variable global */

int main (void)
{
    void lee (void);
    void pinta (void);

    lee ();
    pinta();
}
```

### Fuente 2:

```
extern char c;    /* Aquí se indica que c ya tiene espacio
                    reservado en otro lugar */

void lee (void)
{
    c = getchar();
}
```

### Fuente 3:

```
extern char c;    /* No se reserva espacio para c */

void pinta (void)
{
    putchar (c);
}
```

### 8.1. Introducción.-

Un **array** es una colección de variables del mismo tipo que se referencian por un nombre común. A los elementos del array se accede mediante **índices**.

En memoria, los elementos del array se encuentran almacenados en posiciones contiguas. La dirección más baja corresponderá al primer elemento y la dirección más alta al último elemento del array.

Los arrays pueden tener una **dimensión** o más. Ésta será la principal división que hagamos para estudiarlos. Dentro de los arrays cabe destacar los arrays de caracteres, de gran importancia y en los que se hará especial hincapié.

## 8.2. Arrays unidimensionales.-

La forma general de declaración de un array unidimensional es:

*especificador\_de\_tipo nombre\_variable [tamaño];*

donde *especificador\_de\_tipo* es el tipo base, es decir, el tipo de cada elemento, y *tamaño* es el número de elementos del array.

La forma general de acceder a un elemento del array es:

*nombre\_variable [indice]*

Ejemplo:

```
int vector [5];  
vector [3] = 6;
```

En la primera línea estamos declarando un array de 5 elementos de tipo entero, mientras que en la segunda estamos haciendo uso del cuarto elemento del array para asignarle el valor entero 6. Y estamos accediendo al cuarto elemento del array y no al tercero, ya que en C todos los arrays tienen el cero como índice a su primer elemento. Por tanto, al escribir *int vector [5]* estamos declarando un array de 5 elementos de tipo entero y el array va de *vector[0]* a *vector[4]*.

C no comprueba los límites de los arrays. Esto quiere decir que si escribimos *vector [12] = 7* en el ejemplo anterior, el compilador no me daría ningún tipo de error. Es responsabilidad del programador el indexado correcto de un array. Hay que tener mucho cuidado con esto, pues sin querer, podemos modificar posiciones de memoria no deseadas y el resultado del programa puede no ser el esperado.

### Paso de arrays unidimensionales como parámetros.-

El nombre de la variable de un array es un **puntero** al primer elemento del array. Recordemos que un puntero es la dirección en memoria de esa variable.

Así, cuando se usa un array como un argumento a una función, sólo se pasa la dirección del array, no una copia del array entero. Cuando se llama a una función con un nombre de array, se pasa a la función un puntero al primer elemento del array.

Existen tres formas de declarar un parámetro que va a recibir un puntero a un array. Veamos con un ejemplo las tres formas.

```
#include <stdio.h>

void funcion_ejemplo_1 (int a[10]);
void funcion_ejemplo_2 (int a[]);
void funcion_ejemplo_3 (int *a);

void main (void)
{
    int array [10];
    register int i;

    for (i = 0; i < 10; i++)
        array[i] = i;
    funcion_ejemplo_1 (array);
    funcion_ejemplo_2 (array);
    funcion_ejemplo_3 (array);
}

void funcion_ejemplo_1 (int a[10])
{
    register int i;

    for (i = 0; i < 10; i++)
        printf ("%d", a[i]);
}

void funcion_ejemplo_2 (int a[])
{
    register int y;

    for (i = 0; i < 10; i++)
        printf ("%d", a[i]);
}

void funcion_ejemplo_3 (int *a)
{
    register int y;

    for (i = 0; i < 10; i++)
        printf ("%d", a[i]);
}
```

– En la función *funcion\_ejemplo\_1()*, el parámetro *a* se declara como un array de enteros de diez elementos, el compilador de C automáticamente lo convierte a un puntero a entero. Esto es necesario porque ningún parámetro puede recibir un array de enteros; de esta manera sólo se pasa un puntero a un array. Así, debe haber en las funciones un parámetro de tipo puntero para recibirlo.

En la función *funcion\_ejemplo\_2()*, el parámetro *a* se declara como un array de enteros de tamaño desconocido. Ya que el C no comprueba los límites de los arrays, el tamaño real del array es irrelevante al parámetro (pero no al programa, por supuesto). Además, este método de declaración define *a* como un puntero a entero.

En la función *funcion\_ejemplo\_3()*, el parámetro *a* se declara como un puntero a entero. Esta es la forma más común en los programas escritos profesionalmente en C. Esto se permite porque cualquier puntero se puede indexar usando *[]* como si fuese un array. (En realidad, los arrays y los punteros están muy relacionados). El tema de los

punteros será abordado en el siguiente capítulo, donde se entenderá perfectamente esta última forma de pasar parámetros.

Los tres métodos de declarar un parámetro de tipo array llevan al mismo resultado : un puntero.

### 8.3. Arrays unidimensionales como cadenas de caracteres.-

El uso más común de los arrays unidimensionales es, con mucho, como cadena de caracteres. En C una cadena se define como un array de caracteres que termina en un carácter nulo. Un carácter nulo se especifica como `'\0'`.

Por esta razón, para declarar arrays de caracteres es necesario que sean de un carácter más que la cadena más larga que pueda contener. Por ejemplo, si se desea declarar un array *cad* para contener una cadena de 10 caracteres, se debe escribir:

```
char cad[11];
```

En C, todo lo que esté encerrado entre comillas dobles es una constante de cadena. Por ejemplo:

```
"esto es una cadena"
```

En las constantes de cadenas no es necesario añadir explícitamente el carácter nulo, pues el compilador de C lo hace automáticamente.

### 8.4. Arrays bidimensionales.-

Dentro de los arrays multidimensionales los que más se suelen utilizar son los bidimensionales. Un array bidimensional es, en realidad, un array unidimensional donde cada elemento es otro array unidimensional. Se les suele llamar **matrices**, al igual que a los arrays unidimensionales se les llama **vectores**.

La forma general de declaración es:

```
especificador_de_tipo nombre_variable [tamaño_1] [tamaño_2];
```

y se accede a los elementos del array:

```
nombre_variable [indice_1] [indice_2]
```

El tamaño en bytes de un array bidimensional se calcula con la fórmula:

$$\text{bytes\_de\_memoria} = \text{fila} * \text{columna} * \text{sizeof}(\text{tipo})$$

siendo la declaración del array de la forma :

```
tipo array [fila] [columna];
```

Veamos un ejemplo del uso de arrays bidimensionales :

```
#include <stdio.h>

#define num_filas 4
#define num_columnas 7

void main (void)
{
    int i, j, matriz [num_filas][num_columnas];

    for (i = 0; i < num_filas; i++)
        for (j=0; j<num_columnas; j++)
            matriz[i][j] = i+j;

    for (i=0; i< num_filas; i++)
    {
        for (j=0; j<num_columnas; j++)
            printf ("%2d ", matriz[i][j]);
        putchar ('\n');
    }
}
```

La salida para este ejemplo es:

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9

### Paso de arrays bidimensionales como argumentos a funciones.-

El nombre de un array bidimensional es un puntero al primer elemento del array ([0][0]). Para pasar un array bidimensional como argumento a una función se pasa el puntero al primer elemento. Sin embargo, la función que recibe un array bidimensional como parámetro tiene que definir al menos la longitud de la segunda dimensión.

Veamos el mismo ejemplo de antes, pero ahora usando funciones que utilizan arrays bidimensionales como parámetros.

```
#include <stdio.h>

#define num_filas 4
#define num_columnas 7

void rellenar_matriz (int m[][]);
void imprimir_matriz (int m[][]);

int i, j; /* variables globales */

void main (void)
{
    int matriz [num_filas][num_columnas];

    rellenar_matriz (matriz);
    imprimir_matriz (matriz);
}

void rellenar_matriz (int m[][num_columnas])
{
    for (i = 0; i < num_filas; i++)
        for (j=0; j<num_columnas; j++)
            matriz[i][j] = i+j;
}

void imprimir_matriz (int m[] [num_columnas])
{
    for (i=0; i< num_filas; i++)
    {
        for (j=0; j<num_columnas; j++)
            printf ("%2d ", matriz[i][j]);
        putchar ('\n');
    }
}
```

## 8.5. Arrays multidimensionales.-

En C está permitido el uso de arrays de más de 2 dimensiones. El límite exacto viene determinado por el compilador. La forma general de declaración de un array multidimensional es:

*especificador\_de\_tipo nombre\_array [tam\_1] [tam\_2] ... [tam\_n];*

y la forma de acceder a ellos:

*nombre\_array [ind\_1] [ind\_2] ... [ind\_n]*

Cuando vayamos a pasar arrays multidimensionales como parámetros a funciones, se tiene que declarar todo excepto la primera dimensión.

Por ejemplo, si se declara *arraymult* como:



```
int arraymult [4][3][6][5];
```

entonces la función que reciba *arraymult* podría ser como ésta:

```
void func (int a[][3][6][5])
{
    /* ... */
}
```

## 8.6. Inicialización de arrays.-

La forma general de inicialización de un array es la siguiente:

*especificador\_de\_tipo nombre\_array [tam\_1] ... [tama\_n] = {lista\_de\_valores};*

La *lista\_de\_valores* es una lista de constantes, separadas por comas, cuyo tipo es compatible con *especificador\_de\_tipo*. Después de */* ha de haber un *;*

Veamos un ejemplo de inicialización de un vector:

```
int v[5] = {1, 2, 3, 4, 5};
```

La inicialización de cadenas se puede hacer de dos formas:

```
char cadena[4] = "abc";
char cadena[4] = {'a', 'b', 'c', '\0'};
```

Hay dos formas de inicializar arrays multidimensionales:

```
int m [3][4] =
{
    1, 2, 3, 4,
    5, 6, 7, 8,
    9, 10, 11, 12
};

int m [3][4] =
{
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
}
```

No es necesario que estén todos los elementos en las inicializaciones de arrays. Los elementos que falten se inicializan a 0 o quedan sin valor fijo, según el compilador.

Por ejemplo:

```
int v[5] = {1, 2};
```

En el siguiente ejemplo no se asignará los mismos valores a *m1* que a *m2*:

```
int m1 [3][2] =
{
    2, 3,
    4,
    5, 6
};

int m2 [3][2] =
{
    {2, 3},
    {4},
    {5, 6}
};
```

Sin embargo, una de las formas más comunes de inicialización de arrays es sin especificar el tamaño. Veamos 3 ejemplos:

```
int v[] = {2, 3, 4};

char cadena [] = "esto es una cadena";

int m[][4] =
{
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
```

## 8.7. Ejemplo.-

El siguiente programa implementa un cola estática de caracteres de 10 elementos. En la posición 0 del array se almacena la longitud actual de la cola. Si la cola está llena e intentamos meter más caracteres, la función *meter()* nos devolverá 0.

```
#include <stdio.h>

/* Prototipo de las operaciones sobre la cola */

void crear (char cola[11]); /* cola[0]...cola[10] */
int vacia (char cola[11]);
int meter (char cola[11], char c);
char frente (char cola[11]);
void sacar (char cola[11]);

void crear (char cola[11])
{
    int i;

    for (i=0; i<12; i++)
        cola[i] = 0;
}

int vacia (char cola[11])
{
    if (cola[0] == 0)
        return (1);
    return (0);
}
```

```
int meter (char cola[11], char c)
{
    if (cola[0] < 10)
    {
        ++cola[0];
        cola[cola[0]] = c;
        return (1);
    }
    return (0);
}

char frente (char cola[11])
{
    return cola[1];
}

void sacar (char cola[11])
{
    int i;

    for (i=1; i<11; i++)
        cola[i] = cola[i+1];
    cola[cola[0]] = 0;
    --cola[0];
}

main ()
{
    char cola[11];

    crear (cola);
    meter (cola, 'h');
    meter (cola, 'o');
    meter (cola, 'l');
    meter (cola, 'a');
    while (!vacía(cola))
    {
        putchar (frente(cola));
        Sacar borrar sacar (cola);
    }
}
```



### 9.1. Introducción.-

El concepto de **puntero** es importantísimo en la programación con lenguaje C. Un puntero es una dirección de memoria, normalmente la dirección de alguna variable. En muchos lenguajes, la dirección de las variables sólo es conocida por el compilador, y no puede ser manipulada directamente por el programador. Sin embargo, C no sólo incorpora facilidades de manejo de direcciones, sino que su uso es relativamente frecuente.

A continuación estudiaremos cómo se pueden declarar variables de tipo puntero, cuáles son los operadores asociados y su relación con las funciones.

## 9.2. Variables de tipo puntero y operadores.-

La forma general para declarar un variable puntero es:

*tipo \*nombre;*

donde *tipo* es cualquier tipo válido de C (también llamado tipo base) y *nombre* es el nombre de la variable puntero.

Por ejemplo, las siguientes declaraciones corresponde a 2 punteros, uno al tipo carácter y otro al tipo entero :

```
char *pc;  
int *pe;
```

Existen 2 operadores especiales de punteros: **&** y **\***. Estos dos operadores son monarios y no tienen nada que ver con los operadores binarios de multiplicación (**\***) y de *and* a nivel de bits (**&**).

### El operador &

**&** es un operador monario que devuelve la dirección de memoria de su operando. Veamos un ejemplo:

```
#include <stdio.h>  
  
void main (void)  
{  
    int x = 10;  
    printf (" x = %d\n &x = %p\n", x, &x);  
}
```

Una posible salida para este ejemplo sería la siguiente, teniendo en cuenta que en otras ejecuciones pueden cambiar las direcciones:

```
x = 10  
&x = 8FBC:OFFE
```

### El operador \*

Este operador es el complemento de **&**. Es un operador monario que devuelve el valor de la variable localizada en la dirección apuntada por el puntero que le sigue.

Veamos un par de ejemplos.

## Ejemplo 1:

```
#include <stdio.h>

void main (void)
{
    int x = 10;

    printf (" x = %d\n", x);
    printf (" *x = %d", *x);
}
```

La salida para este ejemplo es:

```
x = 10
*x = 10
```

## Ejemplo 2:

```
#include <stdio.h>

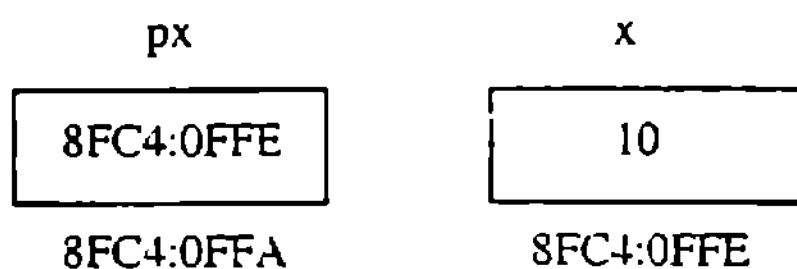
void main (void)
{
    int x = 10;
    int *px;

    px = &x;
    printf (" x = %d\n", x);
    printf (" &x = %p\n", &x);
    printf (" px = %p\n", px);
    printf (" *px = %d", *px);
}
```

La salida para este ejemplo podría ser:

```
x = 10
&x = 8FC4:0FFE
px = 8FC4:0FFE
&px = 8FC4:0FFA
*px = 10
```

Gráficamente:



Como puede verse en este ejemplo, hay tres valores asociados a un puntero:

1. Dirección en la que se encuentra el propio puntero.
2. Dirección a la que apunta el puntero.
3. Valor contenido en la dirección apuntada.

**Nota importante:** las asignaciones siempre han de hacerse en zonas de memoria que hayan sido **reservadas**. Por tanto en el siguiente programa los resultados pueden ser inesperados ya que estamos asignando el valor 10 a una posición de memoria no reservada.

```
void main (void)
{
    int *p;
    *p = 10;
}
```

Sería correcto lo siguiente:

```
void main (void)
{
    int x;          /* se reserva memoria para x */
    int *p;         /* se reserva memoria para el puntero p, no para la
                    /* posición de memoria a la que apunta p */
    p = &x;         /* p apunta a valor de x */
    *p = 10;        /* equivalente a: x = 10 */
}
```

### 9.3. Aritmética de punteros.-

Existen 4 operadores aritméticos que pueden utilizarse con punteros:

+, -, ++, --

Para explicar su uso consideremos el siguiente fragmento de código:

```
int x;
int *px;
px = &x;
```

La expresión  $px + 2$  dará como resultado un puntero a tipo *int*. El valor de la expresión será la dirección de comienzo de *x*, incrementada en dos veces el tamaño de un *int* (es decir,  $2 \times 2$  posiciones de memoria).

En general, si *p* es un puntero al tipo *t* e *i* es un entero, una expresión del tipo  $p+i$ , da lugar a un puntero al tipo *t* cuyo valor es *p* más *i* multiplicado por el tamaño de *t*.

Hay que tener en cuenta que los operadores *contenido* y *dirección* tienen mayor precedencia que los operadores binarios. Comparemos las siguientes expresiones:

```
*p+i    /* Sumar i al valor apuntado por p */
*(p+i)  /* Obtener el valor contenido en la posición apuntada
        /* por (p+i), es decir, desplazar el puntero p i veces,
        /* y a continuación obtener el contenido de esa posición.
        /*
```



## 9.4. Punteros y funciones.-

Como vimos en su momento, los argumentos en C se pasan por valor, es decir, una función sólo trabaja con una copia de los parámetros y no con los originales. Sin embargo hay casos en los que sería deseable que una función pudiera modificar las variables originales, es decir, que se permitiera un paso de parámetros por referencia y no por valor.

Imaginemos que necesitamos una función que intercambie el valor de dos variables de tipo *int*. Para simular el paso de parámetros por referencia hacemos uso de punteros. Cuando invoquemos a la función no pasaremos el nombre de las variables, sino que pasaremos punteros a las variables, con lo cual accederemos directamente a la zona de memoria donde están dichas variables, no a una copia de las mismas. Veamos el ejemplo de intercambio de valores:

```
void intercambio (int *px, int *py)
{
    int temporal;

    temporal = *px;
    *px = *py;
    *py = temporal;
}
```

La forma de utilizar la función *intercambio()* sería similar a ésta:

```
int x=1;
int y=5;
...
intercambio (&x,&y);
```

En el momento de llamar a la función, los parámetros reales se copian en los parámetros formales, es decir, se ejecuta algo semejante a:

```
px = &x;
py = &y;
```

Es importante observar que el paso de parámetros sigue siendo por valor. El valor de los argumentos en el momento de la llamada se copia en *px* y *py*, y una modificación de estas variables (no de las posiciones de memoria a las que apuntan) dentro de *intercambio()* sigue sin tener efecto cuando la función termina de ejecutarse. Considérese el siguiente ejemplo:

```
int main (void)
{
    char *pc;
    void nula (char *pc);
    nula (pc);
}

void nula (char *pc)
{
    char c;
    pc = &c;
}
```

Aquí la función *nula()* no tiene ningún efecto, porque la variable *pc* de dicha función es sólo una copia de la variable *pc* de *main()* y por tanto se destruye al terminar *nula()*.

El ejemplo más usual de paso de punteros como argumentos es la función *scanf()*, de la que ya se ha mencionado algo en capítulos anteriores.

## 9.5. Asignación dinámica de memoria.-

Veamos esta sección con un ejemplo. Supongamos que queremos hacer un programa que lea *n* valores enteros introducidos por teclado por el usuario, los almacene en un vector y los imprima en orden inverso. Una solución podría ser:

```
1 #include <stdio.h>

#define NMAX 100 /* Número máximo de elementos */

void main (void)
{
    int v[NMAX];      /* vector */
    int n = 0;        /* número de elementos introducidos */
    int varaux;        /* variable auxiliar */
    register int i;    /* índice */

    do
    {
        printf ("\nIntroduce número de valores a leer (1-%d): ", NMAX);
        scanf ("%d",&n);
    } while (n < 1 || n > NMAX);
    for (i = 0; i <= n - 1; i++)
    {
        printf ("Introduce valor %d: ", i);
        scanf ("%d", &varaux);
        v[i] = varaux;
    }
    printf ("\n\nValores en orden inverso:\n");
    for (i = n - 1; i >= 0; i--)
        printf ("%d ",v[i]);
}
```

Si el usuario introduce como valor de *n* un 10, estaremos desperdiciando  $90 \times 2$  bytes de memoria, si consideramos que un *int* ocupa 2 bytes. Pero es que además el usuario está limitado a introducir un número por debajo de *NMAX*. Estas restricciones vienen impuestas porque el tamaño de un array en la declaración ha de ser una expresión constante. La asignación de memoria en este caso se dice que es **estática** porque se determina en el momento de la compilación. Cuando la asignación de memoria se determina en tiempo de ejecución se dice que es asignación **dinámica**.

Veamos primero cómo se haría el programa anterior con asignación dinámica y luego pasaremos a explicarlo:

```

#include <stdio.h>
#include <alloc.h>

void main (void)
{
    int *v;          /* vector */
    int n = 0;       /* número de elementos introducidos */
    int varaux;      /* variable auxiliar */
    register int i; /* índice */

    printf ("\nIntroduzca número de valores a leer: ");
    scanf ("%d",&n);
    v = (int *) malloc (n * sizeof (int));
    if (v == NULL)
        printf ("Memoria insuficiente.");
    else
    {
        for (i = 0; i <= n - 1; i++)
        {
            printf ("Introduce valor %d: ", i);
            scanf ("%d",&varaux);
            v[i] = varaux;
        }
        printf ("\n\nValores en orden inverso:\n");
        for (i = n - 1; i >= 0; i--)
            printf ("%d ", v[i]);
        free (v);
    }
}

```

La línea de código que es nueva para nosotros es:

```
v = (int *) malloc (n * sizeof (int));
```

La función *malloc()* reserva memoria. Acepta como argumento los bytes de memoria a reservar y devuelve un puntero al primer byte de la zona de memoria reservada. Los bytes son reservados en un espacio de memoria contiguo. Si no hay suficiente memoria, devuelve NULL (puntero a ningún sitio).

El prototipo de la función *malloc()* es el siguiente:

```
void *malloc (unsigned int bytes);
```

Como vemos devuelve un puntero a cualquier cosa. Nosotros lo que queremos es un puntero a entero, por tanto tendremos que hacer un moldeado con (*int \**)

La memoria asignada no se libera al salir del bloque de código donde fue asignada, como ocurre con las variables locales. Se libera al salir del programa o bien usando la función *free()*. El prototipo de esta función es

```
void free (void *p);
```

El puntero pasado como parámetro ha de ser un puntero para el que se reservó memoria anteriormente con *malloc()*.

Existen otras dos funciones parecidas a *malloc()* en la librería *<alloc.h>*, que son *calloc()* y *realloc()*. Su uso es el siguiente:

```
ptr = (tipo *) calloc (numero, tamaño);
```

```
ptr = (tipo *) realloc (ptr, newsz);
```

*calloc()* espera que se le pase el número de items a crear y el tamaño en bytes de un item. Crea el item, los inicializa a 0 y retorna un puntero al primer byte del bloque completo.

*realloc()* se utiliza para cambiar las dimensiones del área de memoria apuntada por el puntero *ptr* con la dimensión especificada por el parámetro *newsz*, que puede ser mayor o menor que la dimensión original.

## 9.6. Punteros y arrays.-

A estas alturas debe quedar claro que existe una fuerte relación entre los punteros y los arrays. No debemos olvidar que el nombre del array no es sino un puntero al primer elemento del array. Es posible entonces acceder a cualquier array mediante la aritmética de punteros y viceversa, cualquier puntero lo podemos indexar con [].

Por ejemplo:

```
/* Arrays unidimensionales */
p[i] == *(p+i);

/* Arrays bidimensionales */
p[i][j] == *(p+(i*longitud_fila)+k) == (*(p+i)+j);
```

Ejemplo de acceso de un array con un puntero:

```
#include <stdio.h>

void main (void)
{
    float v[3] = {1.1, 2.2, 3.3};
    printf ("v[1] = %g;    *(v+1) = %g", v[1], *(v+1));
}
```

Salida por pantalla:    v[1] = 2.2;    \*(v+1) = 2.2

## Ejemplo de acceso a elementos indexando un puntero con [ ]

```

#include <stdio.h>
#include <alloc.h>

void main (void)
{
    float *p;

    if ((p = (float *) malloc (3 * sizeof (float))) == NULL)
        printf ("\nERROR: Memoria insuficiente.");
    else
    {
        *p = 1.1; *(p+1) = 2.2; *(p+2) = 3.3;
        printf ("*(p+1) = %g;   p[1] = %g", *(p+1), p[1]);
        free(p);
    }
}

```

Salida por pantalla:   \*(p+1) = 2.2;   p[1] = 2.2

Los programadores profesionales de C suelen usar la notación puntero en vez de la notación array por razones de rapidez y comodidad. Cuando trabajemos con cadenas de caracteres debemos utilizar la notación puntero, no ya sólo por eficiencia, sino también por convención.

Una estructura muy común en C es el array de punteros. Recordemos que el argumento *argv* de la función *main()* es un array de punteros a caracteres. Hay 3 formas equivalentes de declarar el argumento *argv* en la función *main()*:

```

main (int argc, char argv[1]);
main (int argc, char *argv[]);
main (int argc, char **argv);

```

Como se podrá observar en la primera declaración no se ha especificado el tamaño de la segunda dimensión de *argv*, cuando dijimos en el anterior capítulo que era necesario. Pues bien, este es el único caso donde se permite, en la función *main()*.

Para finalizar esta sección veamos un último ejemplo donde implementamos 2 funciones que hacen lo mismo, pero la segunda versión es más eficiente. Se trata de buscar un elemento en una matriz bidimensional.

```

#include <stdio.h>
#define N 3

int buscar_en_matriz_1 (int m[N][N], int x)
{
    register int i, j;
    int encontrado = 0;

    for (i = 0; !encontrado && i < N; i++)
        for (j = 0; !encontrado && j < N; j++)
            if (m[i][j] == x)
                encontrado = 1;
    return (encontrado);
}

int buscar_en_matriz_2 (int m[N][N], int x)
{
    register int i;
    int encontrado = 0;
    int *pm = m;

    for (i = 1; !encontrado && i <= N*N; i++)
        if (*pm == x)
            encontrado = 1;
        else pm++;
    return (encontrado);
}

```

## 9.7 Punteros a funciones.-

Esta es una de las herramientas más potentes que nos proporciona el C. Permite pasar una función como argumento de otra función. Con esto se consigue independizar el código de la implementación de la función de la llamada a la función.

Así, por ejemplo, podemos hacer un algoritmo de ordenación que nos valga para ordenar tanto números como letras, dependiendo de la función de comparación que le pasemos como argumento.

De esta manera, la función de ordenación tiene el código del algoritmo de ordenación, pero no el relativo a las comparaciones, que gracias al uso de un puntero a una función de comparación, es independiente de nuestra función.

La sintaxis para declarar una función que lleva como argumento otra función es :

*tipo nombre\_funcion (argumentos,*  
*(tipo \*nombre\_puntero\_a\_funcion)(argumentos) );*

Mientras que la sintaxis de la función pasada como argumento es igual que el de cualquier otra función, sólomente deben coincidir el *número y tipo* de los argumentos y el *tipo* devuelto por la función.

En la librería estándar del C <stdlib> existe la función `qsort()`, que ordena un array siguiendo el algoritmo QSORT. Esta función ordena diferentes tipos de datos, teniendo que pasarle la función de comparación adecuada según cada caso.

Para finalizar, vamos a ver en el siguiente ejemplo una función que compara cadenas de caracteres o números enteros, independiente del tipo de datos que se le pasen, y que por tanto necesita de una función de comparación externa.

Disponemos de dos funciones de comparación, y según el tipo de datos a comparar, le pasamos como argumento la más adecuada.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

/* Prototipos de las funciones */
void comparar (char *a, char *b, int (*cmp)(char *, char *));
int numcmp (char *a, char *b);
int cadcmp (char *a, char *b);

/* Función que compara cadenas o números */
void comparar (char *a, char *b, int (*cmp)(char *, char *))
{
    puts ("\ncomprobando la igualdad\n");
    if ((*cmp) (a,b))
        puts ("\niguales");
    else
        puts ("\n no iguales");
}

/* Función de comparación de números */
int numcmp (char *a, char *b)
{
    puts ("\nestoy en numcmp");
    if (atoi (a) == atoi (b))
        return 0;
    else
        return 1;
}

/* Función de comparación de cadenas */
int cadcmp (char *a, char *b)
{
    puts ("\nestoy en cadcmp");
    return (strcmp (a,b));
}

main ()
{
    char c1[80], c2[80];

    gets (c1);
    gets (c2);
    if (isalpha (*c1))
        comparar (c1,c2,cadcmp);
    else
        comparar (c1,c2,numcmp);
    return 0;
}
```





## *Estructuras, uniones y tipos enumerados*

### 10.1. Introducción.-

Hasta ahora, el único tipo de variable estudiado que nos permitía agrupar varios elementos bajo un solo identificador es el array. Los arrays tienen la limitación de que todos sus elementos son del mismo tipo. En este capítulo estudiaremos otro tipo de variables que nos permite agrupar varios datos de distinto tipo bajo un identificador de variable común. Este tipo de variable es la *estructura*. Para aquel lector que esté familiarizado con el lenguaje PASCAL, una estructura en C (*struct*) es similar a un *record* en PASCAL.

A continuación, estudiaremos las *uniones*, tipo de variables que en principio puede parecer bastante semejante a las estructuras, pero que, como veremos, tienen una semántica muy diferente.

Para finalizar, estudiaremos la definición y utilización de *tipos enumerados* en el lenguaje de programación C.

## 10.2. Estructuras.-

Una estructura es la unión de variables de diferentes tipos en una sola entidad. Vamos a ir viendo cómo se declaran estructuras y qué operaciones se pueden realizar sobre ellas, tanto para el C de K&R como para el ANSI C.

### 10.2.1. Declaración de estructuras

La sintaxis para declarar variables cuyo tipo sea una estructura es la siguiente :

```
struct
{
    especificador_de_tipo_1 nombre_componente_1;
    especificador_de_tipo_2 nombre_componente_2;
    .....
} nombre_variable;
```

De esta forma conseguimos que *nombre\_variable* identifique a una variable cuyo tipo es la estructura que acabamos de definir.

Los componentes de una estructura pueden tener cualquiera de los tipos estudiados para las variables de C, incluso pueden ser otras estructuras. De este modo, podríamos declarar una variable que contuviese los datos de una persona de la siguiente forma :

```
struct
{
    struct
    {
        char nombre [20];
        char apellido_1 [20];
        char apellido_2 [20];
    }nomb_apell;

    struct
    {
        char calle [20];
        char localidad [30];
    }direccion;

    int altura;
}datos _personales;
```

Cada uno de los componentes de la estructura puede ser considerado como una variable diferente. En un sentencia de C, podemos hacer referencia a uno de los componentes de la estructura mediante la siguiente sintaxis :

*nombre\_variable.nombre\_componente\_x*

También podemos declarar arrays de estructuras :

```
struct
{
    float parte_real;
    float parte_imaginaria;
}array_complejos [30];
```

El ámbito del nombre de los componentes de una estructura es precisamente la estructura. De este modo, podemos tener componentes de estructuras con el mismo nombre que variables de nuestro programa o con el mismo nombre que otros componentes de otras estructuras, si bien, esto no se aconseja debido a que normalmente contradice el principio de claridad que hemos de perseguir al escribir nuestros programas.

Podemos declarar más de una variable con la misma estructura de la siguiente forma:

```
struct
{
    float parte_real;
    float parte_imaginaria;
}complejo_1, complejo_2;
```

### 10.2.2. Definición de estructuras

El lenguaje C nos permite definir estructuras independientemente de su declaración. De esta forma, podemos definir primero una estructura y luego, en cualquier punto del programa, podemos declarar una variable que tenga como tipo la estructura definida anteriormente.

La sintaxis para la definición de una estructura es la siguiente :

```
struct nombre_estructura
{
    especificador_de_tipo_1 nombre_componente_1;
    especificador_de_tipo_2 nombre_componente_2;
    .....
};
```

Donde *nombre\_estructura* no es el nombre de una variable cuyo tipo es la estructura, sino el nombre que hemos dado al tipo y que luego podemos asociar a cualquier variable mediante la siguiente sintaxis:

```
struct nombre_estructura nombre_variable_1, nombre_variable_2, .....;
```

De esta forma, podemos definir la estructura :

```
struct COMPLEJOS
{
    float parte_real;
    float parte_imaginaria;
};
```

Y mas tarde, en el programa, realizar la siguiente declaración :

```
struct COMPLEJOS x,y;
```

A partir de este momento, podemos hacer referencia a los componentes de las variables *x* e *y* de la forma acostumbrada:

```
x.parte_real = 3;
y.parte_real = x.parte_real;
.....
```

Hemos de notar que, para diferenciar el modelo de la estructura de las variables declaradas con el tipo estructura, se suelen utilizar mayúsculas para el nombre de la estructura y minúsculas para el nombre de las variables.

### 10.2.3. Operaciones sobre estructuras

#### Operaciones sobre estructuras en el C de Kernighan y Ritchie

Originalmente, en el C descrito por K&R las dos únicas operaciones que podíamos hacer sobre una estructura eran la inicialización de la estructura y la referencia a su dirección de comienzo.

La inicialización de una estructura en el C de K&R se puede realizar siempre que la variable declarada se **global** y **estática**. Además, solo puede realizarse en le momento de la declaración de la siguiente forma :

```
struct COMPLEJO x = {1 , 2};
```

Con lo que conseguimos que *x.parte\_real* sea igual a 1 y *x.parte\_imaginaria* sea igual a 2. Lo que no estaría permitido, por ejemplo, sería:

```
struct COMPLEJO x;
x = {1 , 2};
```

Para hacer referencia a la dirección de comienzo de una variable estructura lo hacemos de la forma acostumbrada. Por ejemplo, podríamos tener la siguiente secuencia de operaciones mediante las cuales declaramos una variable estructura y un puntero a una variable estructura, más tarde, asignamos la dirección de comienzo de la variable estructura al puntero para que así apunte a esta variable:

```
struct NOMBRE_ESTRUCTURA nombre_variable_1;
struct NOMBRE_ESTRUCTURA *nombre_variable_2;
.....
nombre_variable_2 = &nombre_variable_1;
```

Por otro lado, en el C de K&R, no podemos realizar la asignación de una estructura a otra, tendríamos que hacerlo componente a componente. Tampoco podemos pasar estructuras como parámetros en funciones ni éstas pueden devolver estructuras como resultado, aunque esta deficiencia puede ser suplida mediante el uso de punteros a estructuras. La siguiente función compara dos complejos y nos devuelve un nuevo complejo cuya parte real e imaginaria son las menores entre los dos complejos pasados como parámetros:

```
struct COMPLEJO *min_complejo (comp_1, comp_2)
    struct COMPLEJO *comp_1;
    struct COMPLEJO *comp_2;
{
    struct COMPLEJO aux;

    aux.parte_real = ((*comp_1).parte_real < (*comp_2).parte_real) ?
        (*comp_1).parte_real : (*comp_2).parte_real;
    aux.parte_imaginaria = ((*comp_1).parte_imaginaria <
        (*comp_2).parte_imaginaria) ?
        (*comp_1).parte_imaginaria :
        (*comp_2).parte_imaginaria;

    return &aux;
}
```

## Operaciones sobre estructuras en el ANSI C

En el ANSI C, están permitidas todas las operaciones que permitía el C de K&R introduciendo algunas innovaciones, además de añadir otras operaciones más flexibles.

Para comenzar, la inicialización de una variable estructura se puede realizar, no solo cuando ésta es global o estática, sino también cuando se trata de una variable automática.

También podemos asignar una estructura a otra, siempre y cuando éstas sean del mismo tipo :

```
struct COMPLEJO x, y;
.....
x = y;
```

Además, podemos pasar estructuras como parámetros en funciones y devolverlas con la instrucción *return*. De esta forma, nuestra función *min\_complejo* quedaría :

```
struct COMPLEJO min_complejo (comp_1, comp_2)
{
    struct COMPLEJO comp_1;
    struct COMPLEJO comp_2;
    struct COMPLEJO aux;

    aux.parte_real = (comp_1.parte_real < comp_2.parte_real) ?
                    comp_1.parte_real :
                    comp_2.parte_real;
    aux.parte_imaginaria = (comp_1.parte_imaginaria <
                           comp_2.parte_imaginaria) ?
                           comp_1.parte_imaginaria :
                           comp_2.parte_imaginaria;

    return aux;
}
```

Por último, podemos hacer referencia a la dirección de comienzo de cualquiera de los componentes de la estructura :

```
struct COMPLEJO comp;
float *part_imag;
...
part_imag = &comp.parte_imaginaria;
```

Podemos hacer referencia a un componente de una estructura, cuando lo que tenemos es un puntero a ésta, de una forma más cómoda gracias al operador *->* (*guión seguido de un signo menor*) de la forma indicada en el siguiente ejemplo :

```
struct COMPLEJO *comp;
float imag;
...
/* en lugar de imag = (*comp).parte_imaginaria */
imag = comp -> parte_imaginaria;
```

Para finalizar, vamos a ver un largo ejemplo dedicado al tema de punteros y estructuras. El siguiente programa implementa una pila con sus operaciones básicas.

```
#include <alloc.h>
#include <stdio.h>

typedef struct pila_int
{
    int nodo;           /* VALOR ALMACENADO */
    struct pila_int *sig; /* PUNTERO AL NODO SIGUIENTE */
}
pila; /* PILA QUE GUARDA VALORES ENTEROS */

/* OPERACIONES SOBRE LA PILA */
pila *crear (void);
pila *meter (pila *p, int dato);
pila *sacar (pila *p);
```

```

int cima (pila *p);
int vacia (pila *p);

pila *crear (void)
/* CREA LA PILA */
{ return NULL;}

pila *meter (pila *p, int dato)
/* INSERTA UNA NUEVO DATO EN LA PILA */
{
    pila *new_dato;

    new_dato = malloc (sizeof (pila));
    new_dato -> nodo = dato;
    new_dato -> sig = NULL;
    if (p == NULL)
        p = new_dato;
    else
        ( new_dato -> sig = p;
          p = new_dato;
        )
    return (p);
}

pila *sacar (pila *p)
/* BORRA LA CIMA DE LA PILA */
{
    pila *ent;

    ent = p;
    if (p -> sig == NULL)
        p = NULL;
    else
        p = p -> sig;
    free (ent);
    return (p);
}

int cima (pila *p)
/* RETORNA LA CIMA DE LA PILA */
{ return (p -> nodo);}

int vacia (pila *p)
/* RETORNA TRUE (1) SI LA PILA ESTA VACIA */
{
    if (p == NULL)
        return (1);
    return (0);
}

main ()
{
    pila *P;
    int s=0;

    P = crear();
    P = meter (P, 1);
    P = meter (P, 5);
    P = meter (P, 4);
    P = meter (P, 3);
    while (!vacia(P))
    {
        s = s + cima (P);
        P = sacar (P);
    }
    printf ("\n\nsuma = %d", s);
}

```

La ejecución del programa debe dar como resultado en pantalla :  
 suma = 13

### 10.3. Uniones.-

Las uniones, en apariencia, son bastante similares a las estructuras que acabamos de estudiar, pero están dotadas de una semántica muy diferente. De una forma sencilla, mientras las estructuras son conjuntos de variables unidos bajo un sólo identificador, las uniones son una sola variable que puede tomar en cada caso un tipo de valor diferente.

La sintaxis para la definición de una unión es la siguiente :

```
union
{
    especificador_de_tipo_1 nombre_componente_1;
    especificador_de_tipo_2 nombre_componente_2;
    .....
} nombre_variable;
```

De esta forma definimos una variable unión identificada mediante *nombre\_variable* y con los componentes especificados de forma similar a como hacíamos con las estructuras.

La diferencia es que, mientras que en una estructura los componentes de ésta se almacenan en posiciones consecutivas de la memoria, en la uniones, los diferentes componentes, se almacenan en la misma posición; es decir, si mi unión comienza en la dirección *x* de memoria y tiene dos componentes, los dos componentes tienen como dirección de comienzo la *x*. El compilador, para una variable union, sólo reserva el espacio de memoria necesario para almacenar el componente de **mayor** tamaño.

Esto obliga a que, en cada instante, solo pueda ser utilizado uno de los componentes de la unión. En este sentido, podemos observar una variable unión como una variable que, en diferentes instantes, puede comportarse de diferentes formas, tantas como componentes tenga.

Clarifiquemos el concepto de unión mediante un ejemplo. Supongamos que definimos la siguiente unión de dos componentes :

```
union
{
    char letra;
    int número;
}
dia_semana;
```

Con esta definición, pretendemos definir una variable cuyo rango serán los días de la semana, que podremos representarlos bien mediante un entero o bien mediante una letra. De esta forma podríamos realizar las siguiente operaciones :



```

dia_semana.lettra = 'L';
. . . . .
dia_semana.numero = 1;
dia_semana.numero ++;
. . . . .

```

En cada caso, utilizamos *dia\_semana* como una letra o como un número; pero, a diferencia de las estructuras, no podemos utilizar *dia\_semana* como una variable que contiene un número y una letra a la vez.

De este modo, la siguiente secuencia de instrucciones produciría un resultado impredecible :

```

. . . . .
dia_semana.lettra = 'L';
dia_semana.numero ++;
. . . . .

```

Cuando trabajamos con uniones hemos de tomar la precaución de inicializar correctamente la variable cada vez que pasamos a utilizar un nuevo componente. El resto de operaciones sobre uniones se realiza de forma similar a como se realizan sobre estructuras.

## 10.4. Tipos enumerados.-

Los tipos enumerados se utilizan para dar nombre a un conjunto finito de datos y para declarar variables que puedan tomar valores dentro de este conjunto finito. La sintaxis para la definición de un tipo enumerado es la siguiente :

```

enum nombre_tipo_enumerado
{ elemento_1 , elemento_2 , .....};

```

Con esta sintaxis no hemos reservado espacio para ninguna variable, simplemente hemos definido el tipo. Para reservar espacio a una variable cuyo tipo sea *nombre\_tipo\_enumerado* lo haríamos mediante la declaración de la variable:

```

enum nombre_tipo_enumerado nombre_variable_1 [, nombre_variable_2 .....];

```

No es necesario definir un tipo enumerado para declarar una variable con ese tipo. Podemos declararla directamente gracias a la siguiente sintaxis :

```

enum [nombre_tipo_enumerado]
{elemento_1 ,
 elemento_2 , .....} nombre_variable_1, nombre_variable_2 , .....;

```

Nótese que *nombre\_tipo\_enumerado* en este caso es opcional. Cuando no lo incluimos en la declaración decimos que el tipo de las *variables nombre\_variable\_1, nombre\_variable\_2, ...* no tiene nombre.

Un ejemplo de definición de tipo y declaración de variable podría ser el siguiente :

```
enum dias_semana {lunes, martes, miercoles, jueves, viernes,
                  sabado, domingo};
enum dias_semana dia_1, dia_2;
```

Con esta declaración, podríamos realizar las siguiente operaciones :

```
dia_1 = lunes;
dia_2 = martes;
...
if (dia_1 == dia_2)
    ...
}
```

Nótese que no es necesario encerrar los valores del tipo entre comillas para asignarlos a una variable.

El compilador asocia, a cada elemento del tipo enumerado, un valor entero consecutivo, comenzando por el cero. De esta forma, por ejemplo, en nuestra anterior definición, *lunes* tendría el valor 0, *martes* el valor 1 y así sucesivamente. No obstante, nosotros podemos cambiar esta asignación del siguiente modo :

```
enum dias_semana
    {lunes = 5, martes, miercoles, jueves, viernes = 1, sabado, domingo};
```

A partir de este momento *lunes* tendrá asociado el valor 5, *martes* el 6, *miercoles* el 7, *jueves* el 8, *viernes* el 1, *sabado* el 2 y *domingo* el 3. En este sentido, podemos observar los elementos de un tipo enumerado como constantes de nuestro programa a las que se les asigna un valor entero.

A continuación se presenta un ejemplo de función que utiliza el tipo enumerado. La idea es construir una función que dado un día devuelve el siguiente en la semana :

```
enum dias_semana {lunes, martes, miercoles, jueves, viernes,
                  sabado, domingo};

enum dias_semana siguiente_dia (enum dias_semana dia)
{
    enum dias_semana dia_sig;

    switch (dia)
    {
        case lunes :
            dia_sig = martes;
            break;
        case martes :
            dia_sig = miercoles;
            break;
        case miercoles :
            dia_sig = jueves;
            break;
        case jueves :
            dia_sig = viernes;
            break;
    }
}
```

```
        case viernes :
            dia_sig = sabado;
            break;
        case sabado :
            dia_sig = domingo;
            break;
        case domingo :
            dia_sig = lunes;
            break;
    }
    return (dia_sig);
}
```

Podemos implementar esta función de forma más cómoda utilizando el valor entero implícito asociado a cada elemento del tipo por el compilador :

```
enum dias_semana {lunes, martes, miercoles, jueves, viernes,
                  sabado, domingo};

enum dias_semana siguiente_dia (enum dias_semana dia)
{
    return (int) dia + 1;
}
```



## *Entrada / Salida en C. Sistema de ficheros*

### **11.1. Introducción.-**

En el capítulo 4 ya vimos algo relacionado con la entrada y salida de datos en C. Entonces sólo vimos algunas funciones con el fin de hacer un buen seguimiento de los ejemplos que veríamos después. En este capítulo veremos algo más, concretamente ampliaremos nuestros conocimientos de entrada y salida estándar y veremos cómo realizar la E/S en archivos de disco.

## 11.2. Sistemas de E/S.-

En C no existen palabras reservadas para realizar la Entrada/Salida, sino que se lleva a cabo mediante funciones de la biblioteca. Además el C pone a nuestra disposición tres conjuntos distintos de funciones para realizar la E/S:

1. El sistema de E/S definido por el estándar ANSI, también llamado a veces *sistema de archivos con buffer, con formato o de alto nivel*. Estas funciones se encuentran declaradas en la librería `stdio.h`.
2. El sistema de E/S tipo UNIX. Hoy en día bastante en desuso. Fue desarrollado por los primeros compiladores de C. La declaración de las funciones que lo implementan se encuentran ubicadas en la librería `io.h`.
3. Funciones que operan directamente sobre el hardware de la computadora.

En este capítulo sólo hablaremos de las funciones más utilizadas de cada sistema. Si el lector quiere profundizar más puede consultar en los apéndices los apartados dedicados a las librerías `stdio.h` y `io.h`.

## 11.3. Flujos y archivos.-

Cuando un programador se dispone a trabajar con un dispositivo externo, no lo hace directamente con él, sino con una interfaz que el sistema de E/S pone a su disposición. Este interfaz recibe el nombre de **flujo (stream)**. El sistema de archivos de C está diseñado para trabajar con un amplia gama de dispositivos, incluyendo terminales, controladores de disco, etc. Aunque cada dispositivo es diferente, el sistema de archivos con buffer asocia con cada uno de ellos un flujo. Todos los flujos se comportan de la misma manera. Debido a que los flujos son independientes del dispositivo, la misma función puede escribirse en un archivo de disco o en la consola. Por ejemplo, si el usuario hace: `printf("mensaje");` sabe que *mensaje* se escribirá en el flujo estándar de salida, ya sea la pantalla, un fichero de disco, una cinta, etc.

### Tipos de flujo.-

- **Flujos de texto:** son una sucesión de caracteres almacenados en líneas que acaban con un carácter de *nueva línea*. En estos flujos puede que no haya una relación de uno a uno entre los caracteres que son escritos/leídos y los del dispositivo externo. Por ejemplo, una nueva línea puede transformarse en un par de caracteres (*retorno de carro* y carácter de *salto de línea*).
- **Flujos binarios:** son flujos de bytes que tienen una correspondencia uno a uno con los que están almacenados en el dispositivo externo. El número de bytes escritos/leídos es el mismo que el almacenado en el dispositivo externo.

## Archivos.-

En el sistema de E/S del ANSI C, un **archivo** es un concepto lógico que puede ser aplicado por ejemplo tanto a un archivo de disco como a un terminal. Para asociar un flujo con un archivo se tiene que realizar un *apertura* del archivo. A partir de entonces ya podremos intercambiar datos con el archivo a través del flujo. El intercambio de datos se hace por medio de un buffer. Sólo cuando el buffer esté lleno se volcarán los datos hacia el archivo.

## Flujos predefinidos.-

Al comenzarse a ejecutar un programa se abren cinco flujos de texto predefinidos, que se refieren a los dispositivos de E/S estándar conectados al sistema:

<u>Flujo</u>	<u>Dispositivo</u>
stdin	Teclado
stdout	Pantalla
stderr	Pantalla
stdaux	Primer puerto serie
stdprn	Impresora

Los tres primeros flujos están definidos por el C estándar del ANSI y cualquier código que los utilice será totalmente portable. Los dos últimos son específicos de Turbo C y pueden no ser transportables a otros compiladores de C.

Por defecto, la entrada estándar es el teclado y la salida estándar es el monitor. Hay 2 formas básicas de cambiar la entrada y salida estándar:

1. Con los símbolos de redirección del sistema operativo (<, <<, >>) o de tubería (|) al ejecutar el programa desde la línea de comandos del S.O.
2. Con determinadas funciones y variables que se encuentran en la librería <stdio.h> en el código fuente del programa.

## 11.4. E/S por consola.-

Es un subsistema creado dentro del sistema E/S del ANSI, debido a lo comunes que son las entradas por el teclado y las salidas por el monitor. Trabaja con las entrada y salida estándar, por lo tanto pueden redirigirse.

Algunas de estas funciones ya se vieron en su momento en el capítulo 4. Ahora simplemente ampliamos el conjunto.

<u>Función (prototipo)</u>	<u>Operación</u>
int <b>getchar</b> (void)	Lee un carácter del teclado. Espera retorno de carro.
int <b>getche</b> (void)	Lee un carácter con eco. No espera retorno de carro
int <b>getch</b> (void)	Lee un carácter sin eco. No espera retorno de carro.
int <b>putchar</b> (int c)	Escribe un carácter en la pantalla.
char * <b>gets</b> (char *cad)	Lee una cadena del teclado.
char * <b>puts</b> (const char *cad)	Escribe una cadena en la pantalla.

## 11.5. El sistema de archivos de ANSI C.-

Funciones más comunes:

<b>fopen()</b>	Abre un flujo.
<b>fclose()</b>	Cierra un flujo.
<b>putc()</b>	Escribe un carácter en un flujo
<b>getc()</b>	Lee un carácter de un flujo
<b>fseek()</b>	Busca un byte específico de un flujo
<b>fprintf()</b>	Hace lo mismo en flujos que printf en consola
<b>fscanf()</b>	Hace lo mismo en flujos que scanf en consola
<b>feof()</b>	Devuelve cierto si ha llegado al final del archivo
<b>ferror()</b>	Devuelve cierto si se ha producido un error
<b>rewind()</b>	Coloca el localizador de posición del archivo al principio del mismo
<b>remove()</b>	Elimina un archivo

Todo el sistema de E/S con buffer gira en torno a la idea de puntero de archivo. Este será una variable tipo puntero a **FILE**, definido en *<stdio.h>*.

**fopen()** Permite abrir un fichero.

Prototipo **FILE \*fopen (const char \*nombre\_archivo, const char \*modo);**

donde *modo* contendrá el estado de apertura deseado, pudiendo ser:

<u>Modo</u>	<u>Significado</u>
r	Abre un archivo de texto para lectura
w	Crear un archivo de texto para escritura
a	Abre un archivo de texto para añadir
rb	Abre un archivo binario para lectura
wb	Crea un archivo binario para escritura
ab	Crea un archivo binario para añadir
r+	Abre un archivo de texto para lectura/escritura
w+	Crea un archivo de texto para lectura/escritura
a+	Abre o crea un archivo de texto para lectura/escritura
r+b	Abre un archivo binario para lectura/escritura
w+b	Crea un archivo de binario para lectura/escritura
a+b	Abre o crea un archivo binario para lectura/escritura



Ejemplo típico de apertura de fichero:

```
FILE *fp;
...
if ((fp = fopen ("fichero", "w"))==NULL)
{
    puts ("No se puede abrir el archivo\n");
    exit(1);
}
```

**fclose()** Cierra un flujo abierto con `fopen`. Vuelca la información que todavía pueda quedar en el buffer.

Prototipo `int fclose (FILE *pa);`

Devuelve 0 si ha tenido éxito, EOF en caso contrario.

**putc()** Escribe caracteres en un flujo que haya sido abierto previamente para operaciones de escritura usando la función `fopen()`.

Prototipo `int putc (int c, FILE *pa);`

Si tiene éxito devuelve el carácter escrito, si no devuelve EOF.

**getc()** Lee caracteres de un flujo abierto en modo lectura.

Prototipo `int getc (FILE *pa);`

Como se ve, devuelve un entero, pero el byte más significativo es 0. Devuelve EOF cuando se ha alcanzado el final del fichero.

**feof()** Nos dice cuándo un fichero ha alcanzado el final.

Prototipo `int feof (FILE *pa);`

Ejemplo:

```
while (!feof (pa))
    c = getc (pa);
```

**ferror()** Devolverá cierto si se ha producido un error durante la última operación en el fichero `*pa`. El valor de esta función se actualiza en cada operación de fichero.

Prototipo `int ferror (FILE *pa);`

**rewind()** Inicializa el indicador de posición al inicio del archivo indicado por `*pa`.

Prototipo `void rewind (FILE *pa);`

**remove()** Borra el fichero cuyo nombre coincide con *nombre\_fichero*.

Prototipo `int remove (const char *nombre_archivo);`

### **getw() y putw()**

Iguales que `getc()` y `putc()`, pero para enteros. (2 bytes)

**fputs()** Escribe una cadena de caracteres en un flujo que haya sido abierto previamente para operaciones de escritura usando la función `fopen()`.

Prototipo `char *fputs (const char *cad, FILE *pa);`

Si tiene éxito devuelve la cadena de caracteres escrita, si no devuelve EOF.

**fgets()** Lee una cadena de caracteres de un flujo abierto en modo lectura hasta que lee un carácter de nueva línea o de longitud -1.

Prototipo `char *fgets (char *cad, int longitud, FILE *pa);`

Si `fgets()` lee una nueva línea, será parte de la cadena (al contrario que `gets()` ); sin embargo, la cadena resultante será terminada en un nulo.

**fread()** Lee un bloque de datos.

Prototipo `size_t fread (void *buffer, size_t num_bytes, size_t cuenta, FILE *pa);`

**fwrite()** Escribe un bloque de datos.

Prototipo `size_t fwrite (const void *buff, size_t n_bytes, size_t cuenta, FILE *pa);`

**fseek()** Sirve para situar el indicador de posición del archivo. Se recomienda usarlo sólo con ficheros binarios.

Prototipo `int fseek (FILE *pa, long num_bytes, int origen);`

#### Origen

Principio del archivo

Posición actual

Final del archivo

#### Nombre de la macro

SEEK\_SET

SEEK\_CUR

SEEK\_END

**fprintf()** Igual que printf(), pero sobre archivos de disco.

Prototipo `int fprintf (FILE *pa, const char *cadena_fmt,...);`

**fscanf()** Igual que scanf(), pero sobre archivos de disco.

Prototipo `int fscanf (FILE *pa, const char *cadena_fmt,...);`

## 11.6. Rutinas de archivos tipo UNIX.-

- Se mantienen por compatibilidad. Tienden a desaparecer. La mayoría de los compiladores lo soportan.
- Se denomina sin buffer porque el programador es el que debe suministrar y mantener todos los buffer de disco, las rutinas no lo hacen.
- Para el uso de las rutinas necesitamos `<io.h>`

Funciones más comunes:

<code>read()</code>	Lee un buffer de datos.
<code>write()</code>	Escribe un buffer de datos.
<code>open()</code>	Abre un archivo de disco.
<code>close()</code>	Cierra un archivo de disco.
<code>seek()</code>	Va a un byte especificado de un archivo. —
<code>unlink()</code>	Elimina un archivo del directorio.

El manejo de ficheros tipo UNIX gira en torno a lo que se llaman descriptores de fichero, que son de tipo `int`. Antes, en E/S ANSI, teníamos un puntero a `FILE`, ahora tenemos un entero.

**open()** Además de `<io.h>` requiere también `<fcntl.h>`

Prototipo `int open (const char *nombre_archivo, int modo, int acceso);`

El modo puede ser una de las siguientes macros definidas en `<fcntl.h>`:

<u>Modo</u>	<u>Efecto</u>
<code>O_RDONLY</code>	Sólo lectura
<code>O_WRONLY</code>	Sólo escritura
<code>O_RDWR</code>	Lectura/Escritura

A esto se le pueden añadir algunas opciones más.

El parámetro *acceso* sólo se utiliza en entornos UNIX y se incluye por compatibilidad. Generalmente es 0.

Si la llamada es correcta devuelve un entero positivo, caso contrario devuelve -1.

Ejemplo:

```
if ((fd=open (nombre, modo, 0)) == -1)
{
    puts ("No se puede abrir el fichero");
    exit (1);
}
...
```

**close()**            Cierra un archivo.

Prototipo        `int close (int fd);`

Devuelve -1 si no ha sido capaz de cerrar el archivo.

**write()**           Sirve para escribir en un archivo abierto para operaciones de escritura.

Prototipo        `int write (int fd, void *buffer, unsigned num);`

Escribe en el fichero descrito por *fd* un número de bytes indicados por *num* y que se encuentran en *buffer*. Devuelve el número de bytes escritos en el archivo. Si hay algún error devuelve -1.

**read()**            Sirve para leer de un archivo.

Prototipo        `int read (int fd, void *buffer, unsigned num);`

Lee de un fichero descrito por *fd* un número de bytes indicados por *num* y los deja en *buffer*. Devuelve 0 si ha alcanzado el final del fichero y -1 si se ha producido un error. Normalmente devuelve el número de bytes leídos.

**unlink()**           Elimina un archivo del directorio.

Prototipo        `int unlink (const char *nombre_archivo);`

Devuelve -1 si se ha producido un error.

**lseek()** Sirve para hacer un acceso directo a archivo.

**Prototipo** long seek (int *fd*, long *numbytes*, int *origen*);

*numbytes* es el número de bytes que se saltan desde *origen*.

*origen* debe ser una de las siguientes macros:

<u>Origen</u>	<u>Nombre</u>
Principio del archivo	SEEK_SET
Posición actual	SEEK_CUR
Final del archivo	SEEK_END

## 11.7. Ejemplos.-

Para finalizar el capítulo veremos dos ejemplos de manejo de ficheros. El primero emplea las rutinas del ANSI C y el segundo las de UNIX.

### Ejemplo 1 :

```
/* RUTINAS DEL ANSI C */
/* Este programa simula el funcionamiento de la instrucción cat */

#include <stdio.h>
#include <ctype.h>
#include <io.h>
#include <fcntl.h>

filecopy (FILE *fp)
{
    char c;

    while ((c=fgetc (fp)) != EOF)
        fputc (c,stdout);
}

main (ac,av)
int ac;
char **av;
{
    FILE *fp;

    if (ac == 1)
        filecopy (fopen ("con","rw"));
    else
        while (--ac>0)
            if ((fp=fopen(*++av,"r")) == NULL)
            {
                printf ("no puedo abrir %s", *av);
                break;
            }
            else
            {
                filecopy (fp);
                fclose (fp);
            }
}
```

## Ejemplo 2 :

```
/* RUTINAS DEL C DE UNIX */
/* Este programa simula el funcionamiento de la instrucción cat */

#include <stdio.h>
#include <ctype.h>
#include <io.h>
#include <fcntl.h>
#define TAMANO 80

filecopy (int fd)
{
    char c[TAMANO];
    int n;

    while ((n = read(fd,c,TAMANO)) > 0)
        write (1,c,n); /* 1 es la salida estándar */
}

main (ac,av)
int ac;
char **av;
{
    int fd;

    if (ac == 1)
        filecopy (0);
    else
        while (--ac>0)
            if ((fd=open(*++av,O_RDONLY)) == NULL)
            {
                printf ("no puedo abrir %s", *av);
                break;
            }
            else
            {
                filecopy (fd);
                close (fd);
            }
}
```

## *Un ejemplo de programa en C*

### **12.1. Introducción.-**

En este capítulo se presenta un ejemplo completo de programa en C con una envergadura superior a los presentados a lo largo del libro. Con este ejemplo se pretende mostrar el uso combinado de sentencias que, frecuentemente, han de ser utilizadas en nuestros programas.

Para la presentación del ejemplo comenzamos detallando el problema que se pretende abordar, a continuación se presenta el TAD diseñado para la implementación y por último se presenta el código del programa.

## 12.2. Presentación del problema.-

Se va a construir un programa para contener y visualizar los datos de una agenda. En concreto, los datos de la agenda serán gestionados en forma de anotaciones. Una anotación en la agenda vendrá determinada por la fecha y hora en la que ha de surtir efecto la anotación y el texto que nos indique que se ha de hacer en el momento determinado por la fecha y la hora.

El usuario del programa podrá visualizar las distintas anotaciones de la agenda. Las anotaciones se ordenarán en la agenda por fecha y hora de forma ascendente. De esta forma, el programa ofrecerá al usuario la posibilidad de recorrer las distintas anotaciones por orden de fecha y hora tanto hacia delante como hacia detrás. El programa permitirá al usuario la inserción de nuevas anotaciones, la modificación y eliminación de la anotación que actualmente está visualizando.

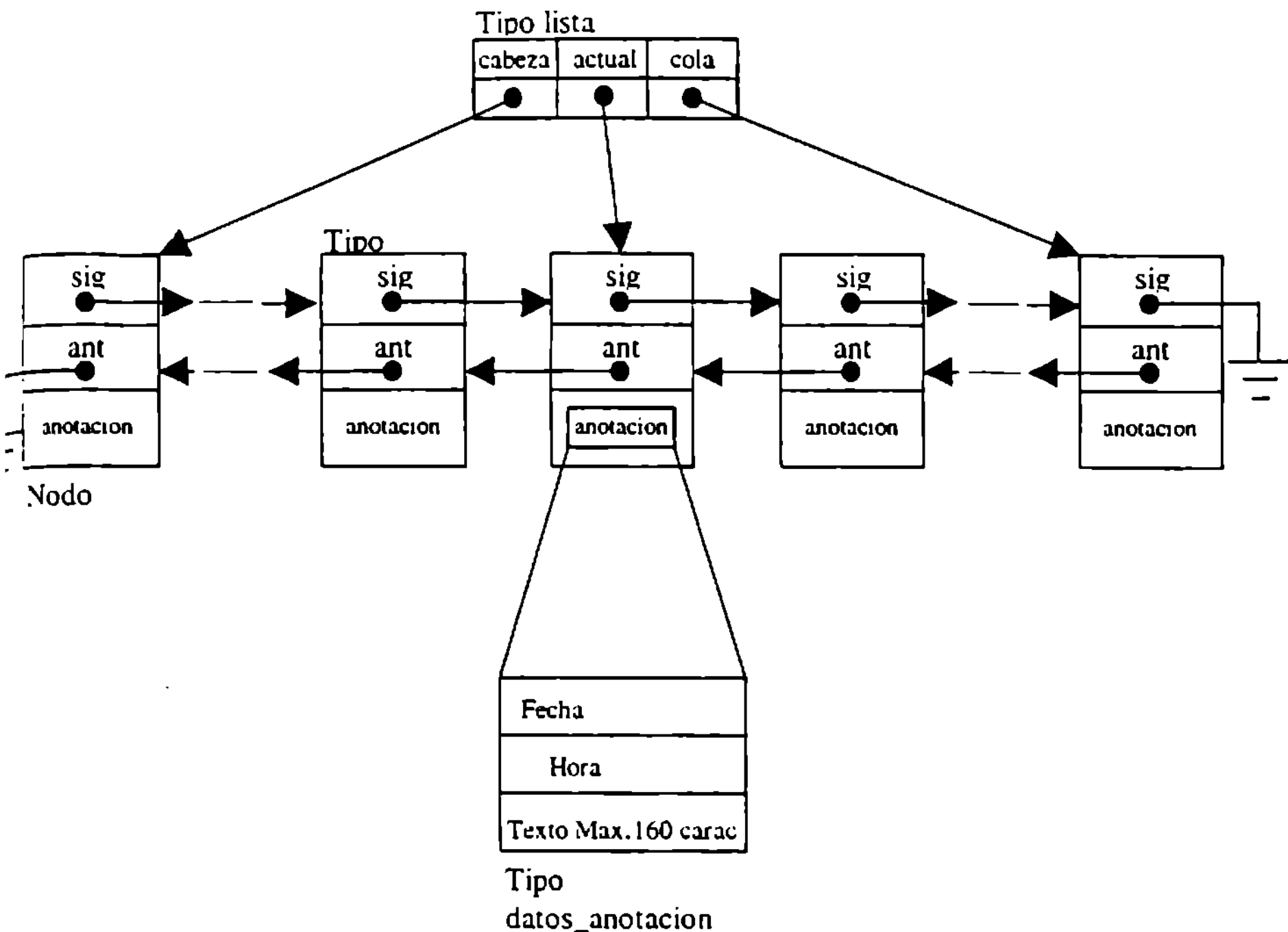
Por último, todas las anotaciones introducidas se mantendrán en memoria principal, pero el programa ofrecerá la posibilidad de volcar las anotaciones de una agenda a un archivo de disco. Del mismo modo, el programa permitirá leer las anotaciones de un archivo para pasarlas a memoria principal. El tipo de archivo elegido es el ANSI C.

Con estas especificaciones del problema vamos a diseñar el TAD para mantener los datos de nuestra agenda.

## 12.3. El TAD.-

Las especificaciones del problema nos piden que los datos de la agenda sean mantenidos en memoria principal, que estén ordenados por un cierto criterio y que puedan ser recorridos en sentido ascendente y descendente pudiendo realizar determinadas operaciones sobre el elemento que actualmente se está visualizando. Así, una estructura adecuada para contener los datos podría ser una lista doblemente enlazada en la que cada nodo contendrá una anotación de la agenda. De este modo, definiremos el tipo *lista* que es una estructura conteniendo tres punteros a datos del tipo *nodo\_lista*, uno de los punteros apuntará a la *cabeza* de la lista, otro a la *cola* de la lista y el restante apuntará a lo que denominaremos el *nodo actual*, es decir, aquel nodo que se está visualizando y sobre el que se ejecutarán en cada momento las diferentes operaciones. El tipo *nodo\_lista* será, a su vez, una estructura que constará de tres campos, los dos primeros serán apuntadores a datos de tipo *nodo\_lista* y se utilizarán para apuntar al nodo siguiente y al nodo anterior en la lista doblemente enlazada; el tercer campo será una estructura que contendrá los datos de cada anotación. Para cada anotación podremos almacenar la fecha, la hora y el texto asociado. La siguiente ilustración muestra gráficamente nuestra estructura de datos.





Hay que hacer notar que la cabeza de la lista apuntará a lo que denominamos un *registro bandera*. Este registro bandera será un registro especial que no contendrá ninguna anotación.

Cuando se crea la lista, se crea una lista vacía con este registro bandera al que apuntan la cabeza, la cola y el nodo actual. Aunque la lista contiene un nodo, será un nodo sin anotación, por lo tanto, semánticamente, se dispone de una lista vacía. La misión de este nodo es la de simplificar los algoritmos de inserción y borrado: se va a implementar la operación de inserción en la lista de forma que la inserción siempre se realice delante del registro actual, si no se dispusiera de este nodo bandera al principio de la lista, la inserción de un nodo delante del primer nodo de la lista necesitaría una operación especial.

De igual forma, la operación de eliminación siempre eliminará el nodo actual, si no se dispusiera de este nodo bandera al principio, la eliminación del primer nodo necesitaría un procedimiento específico.

Las operaciones que completarán nuestro TAD serán :

CONSTRUCTORAS	<p><i>Crear</i> : creará una nueva lista doblemente encadenada con un nodo bandera. Devuelve una lista.</p> <p><i>Insertar(l, a)</i>: inserta en 'l' la anotación 'a' delante del registro actual de 'l'. devuelve una nueva lista.</p> <p><i>Retroceder(l)</i>: hace que el nodo actual de 'l' pase a apuntar a su inmediatamente anterior. Si el nodo actual es el primero, no tiene efecto. Se trata de una operación constructora puesto que la lista no se basa sólo en los datos sino en la posición.</p>
MODIFICADORAS	<p><i>Eliminar(l)</i> : elimina el nodo actual de la lista 'l'.</p> <p><i>Avanzar(l)</i> : Hace que el nodo actual pase a ser el siguiente del actual. Si nodo actual es el último, no tiene efecto.</p> <p><i>Ir_principio(l)</i> : Hace que el nodo actual pase a ser la cabeza de la cola (nodo bandera).</p> <p><i>Ir_primer_nodo(l)</i> : hace que el nodo actual pase a ser el siguiente a la cabeza de la lista.</p> <p><i>Ir_final(l)</i> : hace que el nodo actual pase a ser la cola de la lista.</p>
CONSULTORAS	<p><i>Vacía(l)</i>: devuelve cierto cuando la lista está vacía.</p> <p><i>Final(l)</i>: devuelve cierto cuando el nodo actual es el ultimo de la lista.</p> <p><i>Principio(l)</i> : devuelve cierto cuando el nodo actual es la cabeza de la lista.</p> <p><i>Primer_nodo(l)</i>: devuelve cierto cuando el nodo actual es el siguiente a la cabeza de la lista.</p>

Nótese que se han introducido operaciones que diferencian el hecho de que el nodo actual sea el nodo bandera o el primer nodo después del bandera. Esto se ha hecho así porque en esta implementación consideraremos nodo actual como aquel al que apunta el puntero 'actual' de la lista.

De esta forma se consigue que la implementación C del TAD sea mucho más legible para el lector. La distinción mencionada puede ser obviada si consideramos como nodo actual el nodo siguiente al que apunta el puntero 'actual' de la lista.

En el siguiente punto podemos encontrar el detalle del código correspondiente al programa C que resuelve nuestro problema.

## 12.4. El Código.-

```

/* Programa Agenda */
/* Fecha : 1 de Octubre de 1997 */
/* Autor: Juan Manuel Murillo Rodríguez */
/* Versión: 1.0 */

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>

/*===== ESTRUCTURAS DE DATOS =====*/

typedef struct datos_agenda
{
    char fecha[9];          /* Fecha de la anotación en la agenda */
    char hora[6];           /* Hora de la anotación en la agenda */
    char texto[160];        /* Texto de la anotación */
} datos_anotacion;

typedef struct nodo_lista_d_e /* Nodo de la lista doblemente enlazada */
{
    struct nodo_lista_d_e *sig; /* Puntero al siguiente */
    struct nodo_lista_d_e *ant; /* Puntero al anterior */
    datos_anotacion anotacion; /* Texto de la anotación */
} nodo_lista;

typedef struct lista_d_e /* Definición del tipo lista */
{
    nodo_lista *cabeza; /* La lista se compone de tres elementos: */
    nodo_lista *cola; /* Puntero al principio de la lista, */
    nodo_lista *actual; /* Puntero al final de la lista */
} lista; /* Puntero al nodo actual: sobre el */
/* que se realizan las operaciones. */

/*===== OPERACIONES DEL TAD LISTA =====*/

lista crear (void);
int vacia (lista l);
int final (lista l);
int principio (lista l);
int primer_nodo (lista l);
lista insertar (lista l, datos_anotacion d);
lista eliminar (lista l);
lista avanzar (lista l);
lista retroceder (lista l);
datos_anotacion extraer (lista l);
lista ir_principio (lista l);
lista ir_primer_nodo (lista l);
lista ir_final (lista l);

/*===== OPERACIONES DEL PROGRAMA =====*/

char imprimir_pantalla (lista l);
lista ejecutar_opcion (lista l, char op);
lista opcion_avanzar (lista l);
lista opcion_retroceder (lista l);
lista opcion_eliminar (lista l);
lista opcion_insertar_ord (lista l);

```

```

lista    opcion_modificar    (lista l);
lista    opcion_cargar       (lista l);
void     opcion_salvar        (lista l);
void     convertir_fecha     (char *fecha, char *fecha_sal);
lista    insertar_ord         (lista l, datos_annotacion nueva_annot);

/*===== PROGRAMA PRINCIPAL =====*/

void main (void)
{
    lista l;
    char opcion;

    do
    {
        opcion = imprimir_pantalla (l);
        l = ejecutar_opcion (l, opcion);
    }
    while (!(opcion == 'X') && !(opcion == 'x'));
}

/*===== IMPLEMENTACION DE LAS FUNCIONES DEL TAD =====*/

lista crear (void)
/* Crea un lista doble enlazada. Inicialmente se crea la lista con un
registro bandera. Esto simplifica los algoritmos de inserción y borrado. El
nodo bandera siempre quedará en la cabeza de la lista ya que las inserciones
se realizarán siempre delante del nodo actual */
{
    lista nueva_lista;
    nodo_lista *nuevo_nodo;
    datos_annotacion anot;

    nuevo_nodo = (nodo_lista *) malloc (sizeof(nodo_lista));
    nuevo_nodo -> sig = NULL;
    nuevo_nodo -> ant = NULL;
    nuevo_nodo -> anotacion = anot;
    strcpy(nuevo_nodo->anotacion.fecha, "$");
    strcpy(nuevo_nodo->anotacion.hora, "$");
    strcpy(nuevo_nodo->anotacion.texto, "$");
    nueva_lista.cabeza = nuevo_nodo;
    nueva_lista cola = nuevo_nodo;
    nueva_lista.actual = nuevo_nodo;
    return (nueva_lista);
}

int vacia (lista l)
/* Devuelve true cuando la lista pasada como parámetro esta vacía */
{
    return (l.cabeza == l.cola);
}

int final (lista l)
/* Devuelve true cuando el nodo actual de la lista es el último elemento de
ésta. */
{
    return (l.actual == l.cola);
}

int principio (lista l)
/* Devuelve true cuando el nodo actual es el nodo bandera situado en la
cabeza de la lista */
{
    return (l.actual == l.cabeza);
}

```

```

int primer_nodo (lista l)

/* Devuelve true cuando el nodo actual es el primer nodo de la lista despues
del nodo bandera. */

{return (l.actual == l.cabeza->sig);}

lista insertar (lista l, datos_annotacion d)

/* Inserta un nuevo elemento en la lista. Las inserciones se realizarán
siempre delante del nodo actual. Si no se dispusiera de un nodo bandera en
el inicio de la lista, se tendría que tener una función específica para
realizar inserciones delante del primer nodo de la lista. */

{
    nodo_lista *nuevo_nodo;

    /*Tomamos espacio para un nuevo nodo */
    nuevo_nodo = (nodo_lista *) malloc (sizeof(nodo_lista));
    nuevo_nodo->anotacion = d;
    nuevo_nodo->sig = l.actual->sig;
    nuevo_nodo->ant = l.actual;
    if (! final(l)) /* Si se está al final no tengo nodo siguiente */
        nuevo_nodo->sig->ant = nuevo_nodo;
    else /* Si se ha insertado al final se tiene que modificar */
        /* la cola de la lista. */
        l.cola = nuevo_nodo;
    l.actual->sig = nuevo_nodo;
    l.actual = nuevo_nodo;
    return (l);
}

lista eliminar (lista l)

/* Elimina el nodo actual de la lista. El nodo actual pasa a ser el
siguiente del nodo que se ha eliminado a menos que fuese la cola, en cuyo
caso, el próximo nodo actual será el antecesor del que se acaba de eliminar
*/

{
    nodo_lista *nodo;

    if (! vacia(l)) /* Si la lista esta vacía no se puede eliminar */
    {
        nodo = l.actual;
        l.actual->ant->sig = l.actual->sig;
        if (! final(l))
        {
            l.actual->sig->ant = l.actual->ant;
            l.actual = l.actual->sig;
        }
        else
        {
            l.actual = l.actual->ant;
            l.cola = l.actual;
        }
        nodo->sig = NULL; /* Se libera el espacio ocupado por */
        nodo->ant = NULL; /* el nodo que se acaba de eliminar */
        free (nodo);
    }
    return (l);
}

lista avanzar (lista l)

/* Hace que el nodo actual pase a ser el siguiente del presente nodo actual.
Si el nodo actual es la cola de la lista, no se hace nada. */

{
    if (! final(l))
        l.actual = l.actual->sig;
    return (l);
}

```

```
lista retroceder (lista l)
```

```
/* Hace que el nodo actual pase a ser el antecesor del presente nodo actual.
Si el nodo actual es el nodo bandera, es decir, si nos encontramos al
principio de la lista, no se hace nada. */
```

```
{
    if (! principio (l))
        l.actual = l.actual->ant;
    return (l);
}
```

```
datos_annotacion extraer (lista l)
```

```
/* Devuelve los datos contenidos en el nodo actual. */
```

```
(return (l.actual->anotacion);)
```

```
lista ir_principio (lista l)
```

```
/* Hace que el nodo actual pase a ser el nodo bandera situado al comienzo de
la lista */
```

```
{
    if (! vacia (l))
        l.actual = l.cabeza;
    return (l);
}
```

```
lista ir_primer_nodo (lista l)
```

```
/* Hace que el nodo actual pase a ser el primer nodo de la lista, es decir,
el siguiente al nodo bandera. */
```

```
{
    if (! vacia (l))
        l.actual = l.cabeza->sig;
    return (l);
}
```

```
lista ir_final (lista l)
```

```
/* Hace que el nodo actual pase a ser la cola de la lista */.
```

```
{
    if (! vacia (l))
        l.actual = l.cola;
    return (l);
}
```

```
char imprimir_pantalla (lista l)
```

```
/* Muestra, en caso de que la lista no este vacía, el nodo actual, su
antecesor (si existe) y su sucesor (si existe). Además, muestra el menú de
las diferentes operaciones que se pueden realizar sobre la lista. Devuelve
la opción elegida por el usuario. */
```

```
{
    datos_annotacion aux;
    char opción;
    strcpy(aux.fecha, " ");
    strcpy(aux.hora, " ");
    strcpy(aux.texto, " ");
    if (! (principio(l) || primer_nodo(l)))
    {
        l = retroceder(l);
        aux = extraer(l);
        l = avanzar (l);
    }
    clrscr();
}
```

```

printf("                                VISUALIZACION AGENDA\n");
printf("REGISTRO ANTERIOR :\n");
printf("      Fecha      : %s\n",aux.fecha);
printf("      Hora       : %s\n",aux.hora);
printf("      Texto      : %s\n",aux.texto);
printf("\n\n");
strcpy(aux.fecha," ");
strcpy(aux.hora," ");
strcpy(aux.texto," ");
if (! vacia(l))
    aux = extraer(l);
printf("REGISTRO ACTUAL :\n");
printf("      Fecha      : %s\n",aux.fecha);
printf("      Hora       : %s\n",aux.hora);
printf("      Texto      : %s\n",aux.texto);
printf("\n\n");
strcpy(aux.fecha," ");
strcpy(aux.hora," ");
strcpy(aux.texto," ");
if (! final(l))
{
    l = avanzar(l);
    aux = extraer(l);
    l = retroceder (l);
}
printf("REGISTRO SIGUIENTE :\n");
printf("      Fecha      : %s\n",aux.fecha);
printf("      Hora       : %s\n",aux.hora);
printf("      Texto      : %s\n",aux.texto);
printf("\n\n");
printf("OPCIONES :\n");
printf("      A Avanzar          M Modificar\n");
printf("      R Retroceder      S Salvar\n");
printf("      I Insertar        C Cargar\n");
printf("      E Eliminar        X Salir          Opcion : ");
opcion = getch();
return (opcion);
}

```

lista ejecutar\_opcion (lista l, char op)

/\* En función de la opción elegida por el usuario determina que función o procedimiento ha de ejecutarse. Devuelve la misma lista que recibió como parámetro con las modificaciones que haya sufrido tras la ejecución de la operación. \*/

```

{
    if ((op=='a') || (op=='A'))
        l = opcion_avanzar (l);
    else
        if ((op=='r') || (op=='R'))
            l = opcion_retroceder (l);
        else
            if ((op=='i') || (op=='I'))
                l = opcion_insertar_ord (l);
            else
                if ((op=='e') || (op=='E'))
                    l = opcion_eliminar (l);
                else
                    if ((op=='m') || (op=='M'))
                        l = opcion_modificar (l);
                    else
                        if ((op=='s') || (op=='S'))
                            opcion_salvar (l);
                        else
                            if ((op=='c') || (op=='C'))
                                l = opcion_cargar (l);
                            return (l);
}

```

lista opcion\_avanzar (lista l)

/\* Hace que la lista avance un elemento. \*/

```

{
    l = avanzar(l);
    return (l);
}

```

```
lista opcion_retroceder (lista l)
```

```
/* Permite pasar a la siguiente anotación. */
```

```
{
    if (! primer_nodo(l))
        l = retroceder(l);
    return (l);
}
```

```
lista opcion_eliminar (lista l)
```

```
/* Permite pasar a la notación anterior. */
```

```
{
    l = eliminar(l);
    return (l);
}
```

```
lista opcion_insertar_ord (lista l)
```

```
1 /* Inserta un nuevo nodo en la lista utilizando para ello la función
   insertar_ord. Antes de insertar el nodo pide al usuario que introduzca los
   datos de la nueva anotación. */
```

```
{
    datos_anotacion nueva_anot;

    clrscr();
    printf("INSERCIÓN ORDENADA DE UNA NUEVA ANOTACIÓN.\n\n\n");
    printf("Introduzca los datos de la anotación :\n\n");
    printf("    Fecha de la anotación (DD/MM/AA) : ");
    gets(nueva_anot.fecha);
    printf("    Hora de la anotación (HH:MM)      : ");
    gets(nueva_anot.hora);
    printf("    Texto de la anotación (Max. 160) : ");
    gets(nueva_anot.texto);
    l = insertar_ord (l,nueva_anot);
    return (l);
}
```

```
lista opcion_modificar (lista l)
```

```
/* Esta opción permite modificar los datos pertenecientes a una anotación y
que están contenidos en un nodo. Se permiten modificar todos los datos de la
anotación. Esta función hace uso de funciones de comparación de strings. */
```

```
{
    datos_anotacion nueva_anot,actual;

    clrscr();
    actual = extraer(l);
    l = eliminar(l);
    strcpy(nueva_anot.fecha,"");
    strcpy(nueva_anot.hora,"");
    strcpy(nueva_anot.texto,"");
    printf("MODIFICACIÓN DE LOS DATOS DE LA ANOTACIÓN ACTUAL.\n\n\n");
    printf("Introduzca los datos modificados de la anotación :\n\n");
    printf("    Fecha de la anotación (DD/MM/AA) : [%s] ",actual.fecha);
    gets(nueva_anot.fecha);
    printf("    Hora de la anotación (HH:MM)      : [%s] ",actual.hora);
    gets(nueva_anot.hora);
    printf("    Texto de la anotación (Max. 160) : [%s] ",actual.texto);
    gets(nueva_anot.texto);
    if (0 == strcmp(nueva_anot.fecha,""))
        strcpy (nueva_anot.fecha,actual.fecha);
    if (0 == strcmp(nueva_anot.hora,""))
        strcpy (nueva_anot.hora,actual.hora);
    if (0 == strcmp(nueva_anot.texto,""))
        strcpy (nueva_anot.texto,actual.texto);
    l = insertar_ord (l,nueva_anot);
    return (l);
}
```



```
lista opcion_cargar (lista l)
```

/\* Pide al usuario que introduzca un nombre de archivo. lo abre y lee sus registros insertándolos a continuación en la lista. Para ello en principio vacía la lista. El archivo ha de contener información sobre anotaciones ordenadas por fecha y hora puesto que de lo contrario el funcionamiento del programa no sería correcto. \*/

```
{
    char fichero[13];
    FILE *f;
    datos_anotacion *anot;
    char aux;

    clrscr();
    printf ("Introduzca el nombre del que se quiere cargar : ");
    scanf ("%s",fichero);
    if ((f = fopen(fichero,"rb")) == NULL)
    {
        printf("El archivo no pudo abrirse. Pulse return para continuar.");
        aux = getche();
    }
    else
    {
        while (! vacia(l))
        {
            l = eliminar (l);
        }
        fread(anot,sizeof(datos_anotacion),1,f);
        while (! feof(f))
        {
            l = insertar(l,*anot);
            fread(anot,sizeof(datos_anotacion),1,f);
        }
        l = ir_primer_nodo(l);
        fclose(f);
    }
    return (l);
}
```

```
void opcion_salvar (lista l)
```

/\* Salva los datos de las anotaciones de la lista actual en un archivo. Para ello, inicialmente se pide al usuario que introduzca el nombre del archivo en el que desea volcar la información. \*/

```
{
    char fichero[13];
    FILE *f;
    datos_anotacion *anot;
    char aux;

    clrscr();
    if (! vacia (l))
    {
        printf ("Introducir el archivo sobre el que se quiere salvar: ");
        gets (fichero);
        if ((f = fopen(fichero,"wb")) == NULL)
        {
            printf("El archivo no pudo abrirse. Pulse para continuar.");
            aux = getche();
        }
        else
        {
            l = ir_primer_nodo(l);
            do
            {
                *anot = extraer(l);
                fwrite(anot,sizeof(datos_anotacion),1,f);
                l = avanzar(l);
            }
            while (! final(l));
            *anot = extraer(l);
            fwrite(anot,sizeof(datos_anotacion),1,f);
            fclose(f);
        }
    }
}
```

```

void convertir_fecha (char *fecha,char *fecha_sal)
{
    char aux[7];

    aux[5] = fecha[8];
    aux[0] = fecha[6];
    aux[1] = fecha[7];
    aux[2] = fecha[3];
    aux[3] = fecha[4];
    aux[4] = fecha[0];
    aux[5] = fecha[1];
    strcpy (fecha_sal,aux);
}

```

```

lista insertar_ord (lista l,datos_anotacion nueva_anot)

```

/\* Inserta un nuevo nodo en la lista. La inserción se hará de forma ordenada atendiendo a la fecha de la nueva anotación. Puesto que todas las inserciones se harán con esta función conseguiremos una lista donde todos los nodos estarán ordenados por fecha y hora. Lo primero que realiza la función es pedir al usuario que introduzca los datos de la anotación a insertar. \*/

```

{
    datos_anotacion actual;
    char factual[7];
    char fininsertar[7];
    char hininsertar[7];
    char hactual[7];
    long lfininsertar,lfactual,lhininsertar,lhactual;

    convertir_fecha (nueva_anot.fecha,fininsertar);
    fininsertar[2] = '0';
    lfininsertar = atol(fininsertar);
    strcpy(hininsertar,nueva_anot.hora);
    hininsertar[2] = '0';
    lhininsertar = atol(hininsertar);
    if (vacía(l))
        l = insertar(l,nueva_anot);
    else
    {
        l = ir_principio (l);
        do
        {
            /* Buscamos el lugar correcto para */
            /* la insercion ordenada */
            l = avanzar(l);—
            actual = extraer (l);
            convertir_fecha(actual.fecha,factual);
            lfactual = atol(factual);
            strcpy(hactual,actual.hora);
            hactual[2] = '0';
            lhactual = atol(hactual);

        }
        while (((lfininsertar > lfactual) ||
            ((lfininsertar == lfactual) && (lhininsertar > lhactual))) &&
            (! final (l)));
        if ((lfininsertar < lfactual) ||
            ((lfininsertar == lfactual) && (lhininsertar < lhactual)))
            l = retroceder (l);
        l = insertar (l,nueva_anot);
    }
    return (l);
}

```

## 12.5. Conclusión.-

Con este ejemplo se ha pretendido mostrar el uso de:

Cadenas de caracteres

Arrays

Punteros

Archivos

T.A.D.

Programación funcional

Es decir, lo mínimo que se ha de saber por haber completado nuestra excursión por la Introducción al lenguaje C.



## *El preprocesador y las opciones de compilación*

### 13.1. Introducción.-

Dentro del código fuente de un programa C podemos incluir diversas instrucciones para el compilador. Estas instrucciones reciben el nombre de **directivas de compilación**, y son interpretadas por el preprocesador, justo antes de la compilación del código fuente. El preprocesador realiza el primer análisis del código fuente, llevando a cabo las acciones indicadas en cada caso por las directivas de compilación.

Las directivas de compilación podemos agruparlas en diferentes categorías atendiendo al tipo de uso que tienen. Así, podemos encontrar directivas que condicionan la compilación de una cierta parte del código fuente dependiendo de un cierto valor lógico; también nos sirven para definir macros y constantes, o para poder incluir código ensamblador dentro de un programa escrito en C.

## 13.2. Preprocesado. Directivas de compilación.-

De esta forma, podemos considerar que el preprocesado es una primera pasada sobre el código fuente en la que todas las directivas de compilación se sustituyen por su significado, de tal manera que cuando actúe el compilador no se va a encontrar más que código C.

Las directivas de compilación se diferencian del resto de instrucciones por su apariencia: siempre comienzan con el símbolo (#), y nunca terminan en punto y coma (;). Las principales directivas de compilación definidas para el preprocesador de C son :

Directiva de inclusión :	#include
Directivas de definición :	#define, #undef
Directivas de depuración :	#error, #line
Directivas condicionales :	#ifdef, #ifndef, #if, #elif, #else, #endif
Otras directivas :	#pragma

## 13.3. Directiva de inclusión.-

La directiva **#include** indica al preprocesador que debe incluir otro archivo fuente. El archivo a incluir debe ir encerrado entre paréntesis angulares ( < > ) o entre comillas dobles ( " " ), según sea el caso :

1. Irá entre paréntesis angulares cuando el archivo se encuentre almacenado en el directorio de ficheros incluidos, definido por el propio entorno del C. Normalmente se emplea para los archivos que aporta el C en sus librerías estándar.
2. Por el contrario, si el fichero a incluir se encuentra en el directorio actual de trabajo, debemos indicar el nombre del fichero entre comillas dobles. Se suele emplear para los archivos definidos por el usuario.

En cualquier caso, se puede indicar el *path* completo de búsqueda. Vamos a verlo a través de un ejemplo :

Supongamos la siguiente jerarquía de directorios :

C:\TC	subdirectorio de turbo C
C:\TC\INCLUDE	subdirectorio de ficheros incluidos
C:\TC\LIB	subdirectorio de librerías
C:\TC\TRABAJO	subdirectorio actual de trabajo
.....	

Entonces, podíamos tener el siguiente programa ejemplo :

```
#include <stdio.h>
/* el compilador buscará el archivo stdio.h en C:\TC\INCLUDE */

#include "listas.h"
/* el compilador buscará el archivo listas.h en C:\TC\TRABAJO */

#include "c:\cosas\pepe.h"
/* el compilador buscará el archivo pepe.h en C:\COSAS */
....
```

El efecto de una directiva de inclusión es que el preprocesador sustituye la directiva *#include <fichero>* por el fichero especificado, es decir, literalmente se copia encima. Esta es la razón de porqué no se puede incluir más de una vez un cierto fichero en un programa C: cuando el compilador actuase, se encontraría duplicado el código correspondiente al fichero incluido.

### 13.4. Directivas de definición.-

#### La directiva *#define*

Esta directiva se emplea para definir macros de la forma :

*#define nombre valor*

De tal manera que el preprocesador sustituirá cada ocurrencia de *nombre* por su *valor* justo antes de la compilación del código fuente.

La definición de macros en C es bastante potente, aunque normalmente la directiva *define* se emplea para definir constantes, por ejemplo :

```
#define FALSE 0
#define TRUE !FALSE
```

De esta forma hemos definido dos constantes que podemos usar a lo largo de nuestro programa. En el momento del preprocesado, cada ocurrencia de *FALSE* se sustituirá por *0*, y cada ocurrencia de *TRUE* por *!0*.

En este sentido, los nombres de macro se suelen escribir con letras mayúsculas. Esta convención ayuda a quien lee el programa.

El porqué no se emplea punto y coma (;) al final de la directiva, podemos verlo en el siguiente ejemplo :

```
#define VERSION "versión 2.0"

printf (VERSION);
```

Si hubiera acabado en punto y coma (;), después del preprocesado nos hubiera quedado algo como :

```
printf ("versión 2.0");
```

con el consiguiente error de sintaxis.

Si queremos construir una macro que se sustituya por una cadena tan larga que no quepa en una sola línea, haciendo uso de “\” se puede extender a líneas sucesivas. Vamos a verlo en el siguiente ejemplo :

```
#define CADENA "hola esto es una cadena muy larga \
que no cabe una sola línea."

printf ("CADENA LARGA : \n");
printf (CADENA);
```

Daríamos como salida :

```

|
CADENA LARGA :
hola esto es una cadena muy larga que no cabe en una sola línea.
```

Con esto vemos que los nombres de macro no se sustituyen por su valor si van entre comillas dobles, porque el preprocesador asume que forma parte de una cadena.

El siguiente uso que vamos a ver de la directiva *#define* es para construir una macro con argumentos. Estos argumentos no necesitan definirse de un cierto tipo, el preprocesador asocia el argumento con su valor por la posición que ocupa. Vamos a verlo en el siguiente ejemplo :

```
#define SUMA (a,b) (a+b)      --
#define RESTA (x,y) x-y
```

Si en nuestro programa escribimos algo como :

```
main ()
{
    int a,z ;

    a = SUMA (5,3) * 2;
    z = RESTA (5,3) * 2;
}
```

Esto será sustituido en el preprocesado por :

```
main ()
{
    int a,z ;

    a = (5+3) * 2;
    z = 5-3 * 2;
}
```



Según el ejemplo, vemos la conveniencia de usar paréntesis en las definiciones de las macros para evitar problemas cuando el preprocesador sustituya el nombre por su valor. En el segundo caso, a z no se le asigna el valor deseado.

También podemos anidar macros :

```
a = SUMA ( SUMA (1,6), RESTA (SUMA (1,7), 5) )
```

En cuyo caso, el empleo de paréntesis es casi imprescindible para que el resultado sea el deseado. Veamos otro ejemplo :

```
#include <stdio.h>

#define MIN (a,b)  ( (a) < (b) ) ? (a) : (b)

main ()
{
    int z, x=10, y=20;

    z = MIN (x,y);
    printf ("El mínimo es :   %d\n\n", z);
    z = MIN (x+5, y-7);
    printf ("y ahora es   :   %d\n\n", z);
}
```

Esto será preprocesado y sustituido por :

```
....
main ()
{
    int z, x=10, y=20;

    z = ( (x) < (y) ) ? (x) : (y) ;
    printf ("El mínimo es :   %d\n\n", z);
    z = ( (x+5) < (y-7) ) ? (x+5) : (y-7) ;
    printf ("y ahora es   :   %d\n\n", z);
}
```

A primera vista podíamos pensar que sobran algunos paréntesis, pero en el segundo caso queda claro que son necesarios todos.

Si el valor de sustitución de la macro engloba varias instrucciones, hay que emplear llaves ( { } ). Ejemplo :

```
#define INTERCAMBIO (x,y)  { int tmp; tmp = x; x = y; y = tmp; }
```

o lo que es lo mismo :

```
#define INTERCAMBIO (x,y)  { int tmp; \
                             tmp = x; \
                             x = y; \
                             y = tmp; }
```

También podemos encontrar sucesivas aplicaciones de macros, por ejemplo :

```
#define SUMAR (x,y)    x+y
#define MAT (op,x,y)   op (x,y)

main ()
{
    int z;

    z = MAT (SUMAR, 3, 4);
}
```

Esto será preprocesado y sustituido primeramente por :

```
....
z = SUMAR (3,4);
```

y a continuación se vuelve a preprocesar, dando como resultado final :

```

}
....
z = 3+4;
```

Finalmente para terminar con esta directiva vamos a ver lo que se conoce con el nombre de **caracteres de expansión de macro**. Son dos :

- # Sustituye lo que va a continuación por eso mismo entre comillas dobles.
- ## Sustituye lo que va a continuación por eso mismo, dando lugar a un nuevo símbolo.

Ejemplo :

```
#define MOSTRAR (x) printf ("la variable \"%x\" vale %d.\n", x)
#define VER (x)      printf ("la variable__man vale %d.\n", m##xn)

main
{
    int a=10, man=20;

    MOSTRAR (a);
    VER (a);
}
```

La macro MOSTRAR será sustituida por :

```
printf ("la variable \"%a\" vale %d.\n", a);
```

En pantalla aparecerá :

la variable a vale 10.

La macro VER será sustituida por :

```
printf (la variable man vale %d.\n", man);
```

En pantalla aparecerá :

la variable man vale 20.

En el caso de VER, si la variable *man* no existiera, se produciría un error durante la compilación del código.

Veamos otro ejemplo :

```
#define Sub 2
#define SH(x) printf ("n°%x" = %d ó %d.\n", n°#x, alt(x))

main ()
{
    int nSub=3. alt(5)=(2,5,1,3,6);

    SH (Sub);
}
```

Vamos a ver paso a paso el preprocesado de esta macro :

```
printf ("n°"Sub"" = %d ó %d.\n", nSub, alt[Sub]); /* paso 1 */
printf ("nSub = %d ó %d.\n", nSub, alt(Sub));      /* paso 2 */
printf ("nSub = %d ó %d.\n", nSub, alt[2]);        /* paso 3 */
```

Y en pantalla aparecerá :                      nSub = 3 ó 1.

En general, las macros no son muy usadas porque hacen crecer al código, ya que en el preprocesado se sustituye el nombre de la macro por su valor. Además, el control de errores es mucho más complejo (no hay chequeo de tipos, por ejemplo).

Sin embargo, no todo son desventajas. La llamada a una macro es más rápida que la llamada a una función, precisamente porque después del preprocesado la llamada a macro se convierte en una línea de código más (ya no hay salto en el flujo de programa).

## La directiva *#undef*

Esta directiva se usa para quitar una definición de macro que anteriormente se había declarado. El formato general es :

*#undef nombre\_de\_macro*

Ejemplo :

```
#define ALTO 100
#define ANCHO 40

char matriz[ALTO][ANCHO];

#undef ALTO
#undef ANCHO
/* en este momento ALTO y ANCHO están indefinidos, ya no se pueden usar */
```

El uso de esta directiva permite restringir el uso de los nombres de macros a aquellas secciones de código en las que se necesiten.

## 13.5. Directivas de depuración.-

### La directiva *#error*

Esta directiva se emplea para forzar al compilador a parar cuando se la encuentre. Se usa principalmente para depuración. Su formato general es :

*#error mensaje*

*mensaje* no es necesario que vaya entre comillas dobles. Cuando el compilador encuentra esta directiva, visualiza la siguiente información y termina la compilación. Por ejemplo :

```
....
#error He compilado hasta la línea 125.
```

Cuando el compilador llegue a esta directiva, interrumpe la compilación y muestra el mensaje.

### La directiva *#line*

Al igual que la anterior, esta directiva se suele usar para depurar programas. *#line* permite cambiar los contenidos de `__LINE__` y `__FILE__`, dos identificadores predefinidos en Turbo C.

`__LINE__` almacena el número de línea en la que se encuentra una cierta instrucción dentro del código fuente. `__FILE__` almacena el nombre del fichero donde se encuentra el código a compilar. El formato general de la directiva *#line* es :

*#line número nombre\_archivo*

donde, *número* es el nuevo valor que se le da a la siguiente línea de código, y *nombre\_archivo* es cualquier identificador válido que represente el nuevo nombre del archivo.

Esta directiva sólo tiene sentido para el compilador de línea de comandos, ya que el entorno integrado de Turbo C la ignora.

En el siguiente ejemplo, se modifica el número de línea para que comience por la línea 100 :

```
#include <stdio.h>

#line 100
main ()                /* línea 100 */
{                      /* línea 101 */
    printf ("%d\n", __LINE__); /* línea 102 */
}
```

La sentencia *printf* visualizará :

102

## 13.6. Directivas condicionales.-

### Las directivas *#ifdef* , *#ifndef*

Podíamos traducirlas por “*definido*” , “*no definido*”. En función de que anteriormente se haya definido o no definido cierto nombre de macro, se compila o no cierto código fuente. El formato general es :

```
#ifdef nombre_de_macro
    secuencia de sentencias
#endif
```

```
#ifndef nombre_de_macro
    secuencia de sentencias
#endif
```

En el caso de *#ifdef*, si el nombre de macro ha sido definido se compilará la secuencia de sentencias entre *#ifdef* y *#endif*.

Para el caso de *#ifndef*, la secuencia de sentencias se compilará en caso de no haberse definido el nombre de macro.

Se permite anidar tanto el *#ifdef* como el *#ifndef*.

Ejemplo :

```
#include <stdio.h>
#define PEPE 10

main
{
    #ifdef PEPE
        printf ("Hola Pepe\n");
    #endif

    # ifndef JUAN
        printf ("Juan no está definido");
    #endif
}
```

Este programita dará como salida en pantalla :

```
Hola Pepe
Juan no está definido
```

### Las directivas *#if*, *#else*, *#elif*, *#endif*

Permiten compilar una secuencia de sentencias en función de una cierta condición. El formato más general es :

```

    #if expresión_constante
        secuencia de sentencias
    [#elif expresión 1
        secuencia de sentencias]
    ....
    [#elif expresión N
        secuencia de sentencias]
    [#else
        secuencia de sentencias]
    #endif

```

Las expresiones entre corchetes ( [ ] ) indican que esa parte es opcional.

Si la *expresión\_constante* se evalúa como verdadera, se compila la siguiente secuencia de sentencias hasta llegar al *#endif* o hasta que se encuentre una de las expresiones opcionales.

```

/* ejemplo simple de #if */

#include <stdio.h>
#define MAX 100

main ()
{
    #if MAX > 99
        printf ("Compilado para un valor mas grande que 99\n");
    #endif
}

```

*#elif* es similar a "else if " y se usa para establecer una cascada *if/else/if* para opciones de compilación múltiple. Sólo se compila la secuencia de sentencias asociada a aquella expresión que se evalúe como verdadera.

```

/* ejemplo de cascada */

#define USA 0
#define ESP 1
#define FRA 2

#define PAIS USA

#if PAIS == USA
    char moneda[] = "dólar";
#elif PAIS == ESP
    char moneda[] = "peseta";
#elif PAIS == FRA
    char moneda[] = "franco";
#endif

```

Finalmente, la secuencia de sentencias asociadas al *#else* se compilan cuando ninguna de las opciones anteriores se haya evaluado como verdadera.

```
/* ejemplo uso else */

#include <stdio.h>
#define MAX 10

main ()
{
    #if MAX > 99
        printf ("Compilado para un valor grande\n");
    #else
        printf ("Compilado para un valor pequeño\n");
    #endif
}
```

Se permite anidar directivas de compilación condicional como las que acabamos de ver hasta cualquier nivel. En el caso de las directivas *#ifdef* , *#ifndef* sólo se permite anidarlas con las directivas condicionales *#if* , *#else*.

```
/* ejemplo anidado */

#define MAX 100
#define VERSION 2.5

main ()
{
    #if MAX == 100
        #if VERSION == 2.0
            printf ("Opción 1 activada\n");
        #elif VERSION < 2.0
            printf ("Opción 2 activada\n");
        #else
            printf ("Opción 3 activada\n");
        #else
            #ifdef VERSION
                printf ("Opción 4 activada\n");
            #endif
        #endif
    }
}
```

### 13.7. Otras directivas.-

En realidad solamente vamos a estudiar una directiva, *#pragma*, pero como se emplea para diferentes cosas, es como si fueran 3 directivas.

Permite dar diversas instrucciones, definidas por el creador del compilador, al compilador. En Turbo C se permiten tres instrucciones

#### 1. *#pragma warn especificaciones*

Provoca que Turbo C sobrescriba opciones de mensaje de advertencia. *Especificaciones* es una de las diversas opciones de mensaje de error.

#### 2. *#pragma inline*

Le indica a Turbo C que el programa contiene código ensamblador en línea.

El C es un lenguaje muy potente, pero hay veces que puede interesar escribir algunas instrucciones en ensamblador. Hay tres razones para ello:

- Aumentar la velocidad y la eficiencia.
- Realizar una función específica de la máquina que no esté disponible en C.
- Utilizar una rutina en lenguaje ensamblador empaquetada de propósito general.

Aquí no nos vamos a extender mucho en este tema, sólo nos vamos a limitar a comentar cómo se pueden incluir instrucciones ensamblador en un programa de C (para el caso de Turbo C).

Antes de escribir instrucciones en ensamblador debemos incluir la siguiente directiva:

```
#pragma inline
```

la cual le dice al compilador que el programa contiene estamentos *asm*.

La palabra clave *asm* tiene la siguiente sintaxis:

```
asm codigo_de_operacion operandos ;
```

Ejemplo:

```
int var = 10;
asm mov ax, var
```

Si se quiere incluir varias instrucciones en ensamblador, se crea un bloque (con dos llaves) después de *asm*.

Ejemplo:

```
asm
{
    pop ax; pop ds
    iret
}
```

### 3. *#pragma saveregs*

Se utiliza en el modelo de memoria *huge* para forzar al compilador a salvar todos los registros.

En cualquiera de los tres casos, la directiva *#pragma* debe ir inmediatamente antes de la función en la que se quiera usar, ya que sólo afectará a dicha función. Esta directiva sólo se emplea en casos muy extraños.



## *Introducción a C++: Programación Orientada a Objetos*

### **14.1. Introducción.-**

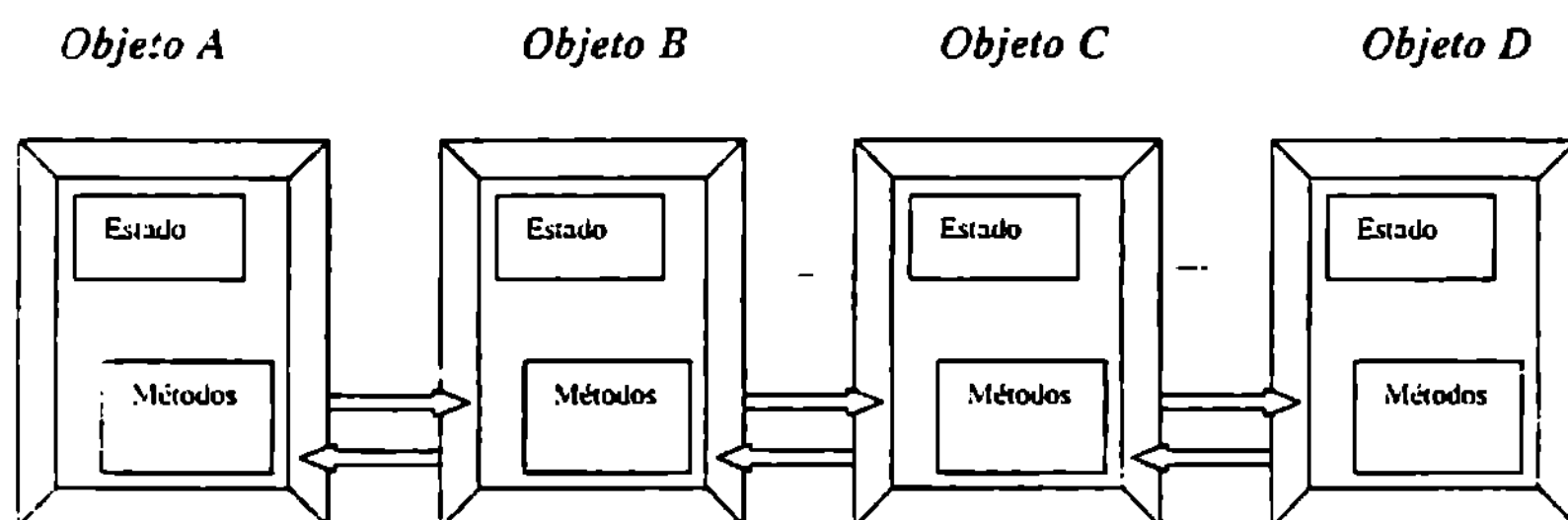
Basándose en la idea natural de un mundo lleno de objetos aparece la Programación Orientada a Objetos (POO) como metodología de programación a finales de los años 60, de manera que la resolución de un problema se realiza en términos de objetos. En la vida real estamos rodeados de objetos que tienen una funcionalidad bien definida y que sabemos aprovechar de manera casi inconsciente.

## 14.2. Conceptos fundamentales.-

### 14.2.1. Clases y Objetos.

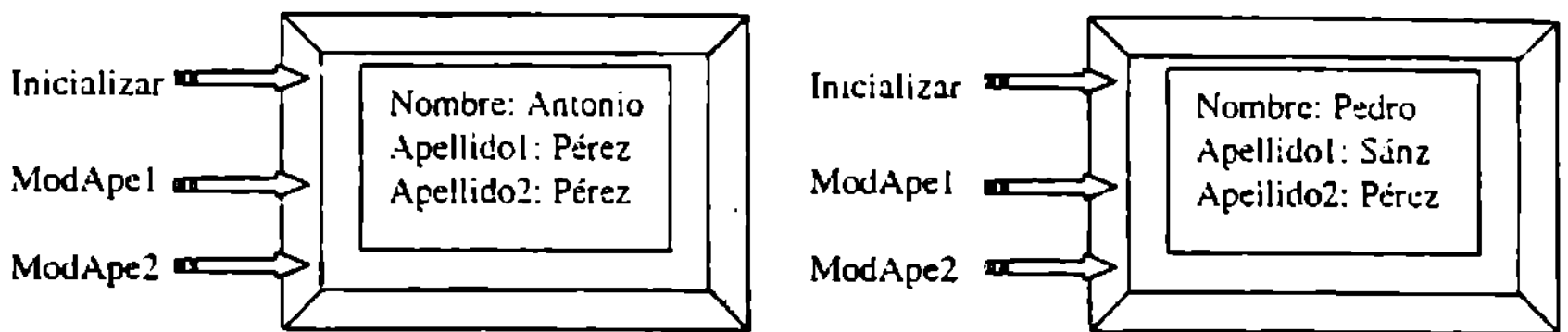
Un *Objeto* es aquella entidad que encapsula los servicios que ofrece a través de un interfaz de paso de mensajes. En la metodología de programación planteada un objeto encapsula un conjunto de datos (atributos) y operaciones (métodos) que manipulan esos datos. De esta manera los datos representan el estado local del objeto y los métodos comparten el estado teniendo en cuenta que la única forma de acceder o modificar el estado de un objeto es a través de los métodos definidos como públicos del mismo. Estos métodos se convierten entonces en la única vía de comunicación de un objeto con el mundo exterior (resto de objetos definidos en el programa).

Para poder definir una variable en cualquier lenguaje de programación, orientado o no a objetos, ésta debe asociarse a un determinado tipo de datos. Cuando se utiliza POO y se desea trabajar con un objeto A éste debe estar asociado a una determinada definición<sup>1</sup> de tipo en el que se especifique los atributos y métodos que se podrán invocar. Para poder implementar esta definición aparecen las *Clases*, de manera que una clase especifica los atributos y métodos, que contendrá y podrá invocar, un determinado objeto. De esta manera se definen los objetos como *Instancias de clase*.



### 14.2.2. Paso de mensajes.

Cuando se conecta una radio sus elementos físicos no preocupan, tampoco su comportamiento (como capta las ondas y las transforma en el sonido deseado), simplemente se enciende y se selecciona la emisora deseada. De alguna manera existe un paso de mensajes, formado en este caso por una serie de acciones bien definidas, hacia al objeto que son interpretados de manera transparente para obtener la funcionalidad deseada. De esta misma manera se puede interpretar la comunicación que debe existir entre objetos dentro de un programa que utilice esta metodología. Así, el mecanismo de paso de mensajes en programación secuencial se simula con la invocación de un método público por parte de una instancia de clase.

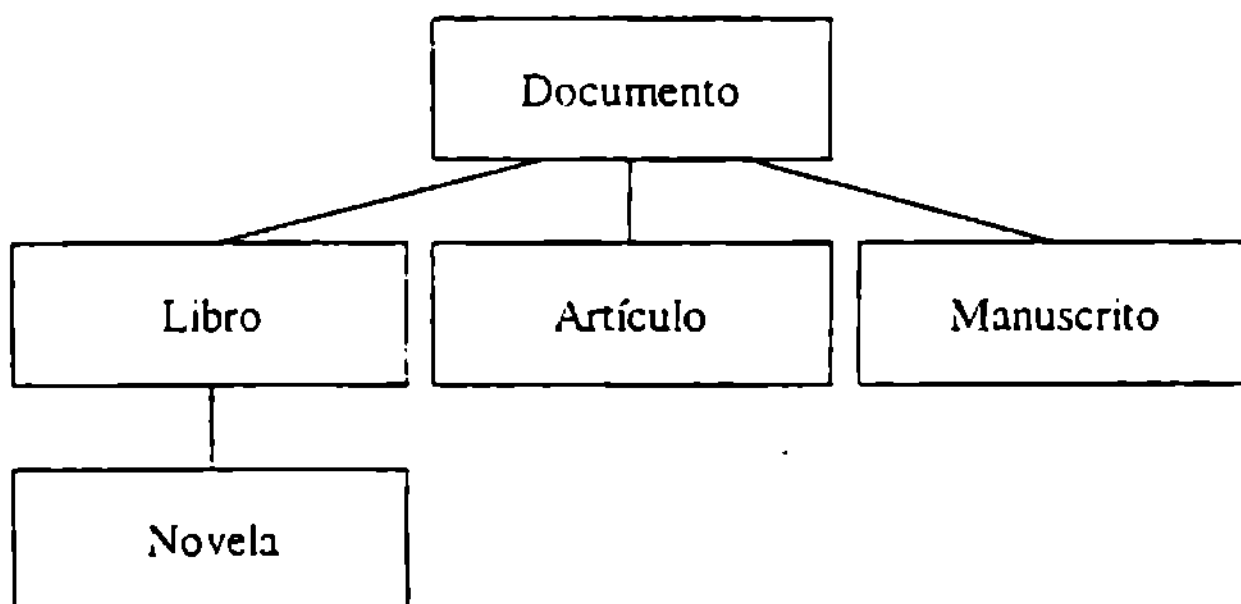


### 14.2.3. Herencia.

La *Herencia* es una de las características más importantes de la POO, permite la reutilización del comportamiento de una clase en la definición de otras denominadas subclases o *clases derivadas* en terminología C++.

De una clase se heredan los métodos y atributos definidos, pudiendo extenderse en la clase derivada para expresar una especialización en el comportamiento de la clase derivada.

Es costumbre expresar el mecanismo de herencia en la metodología orientada a objeto en una carta jerárquica de la siguiente manera:



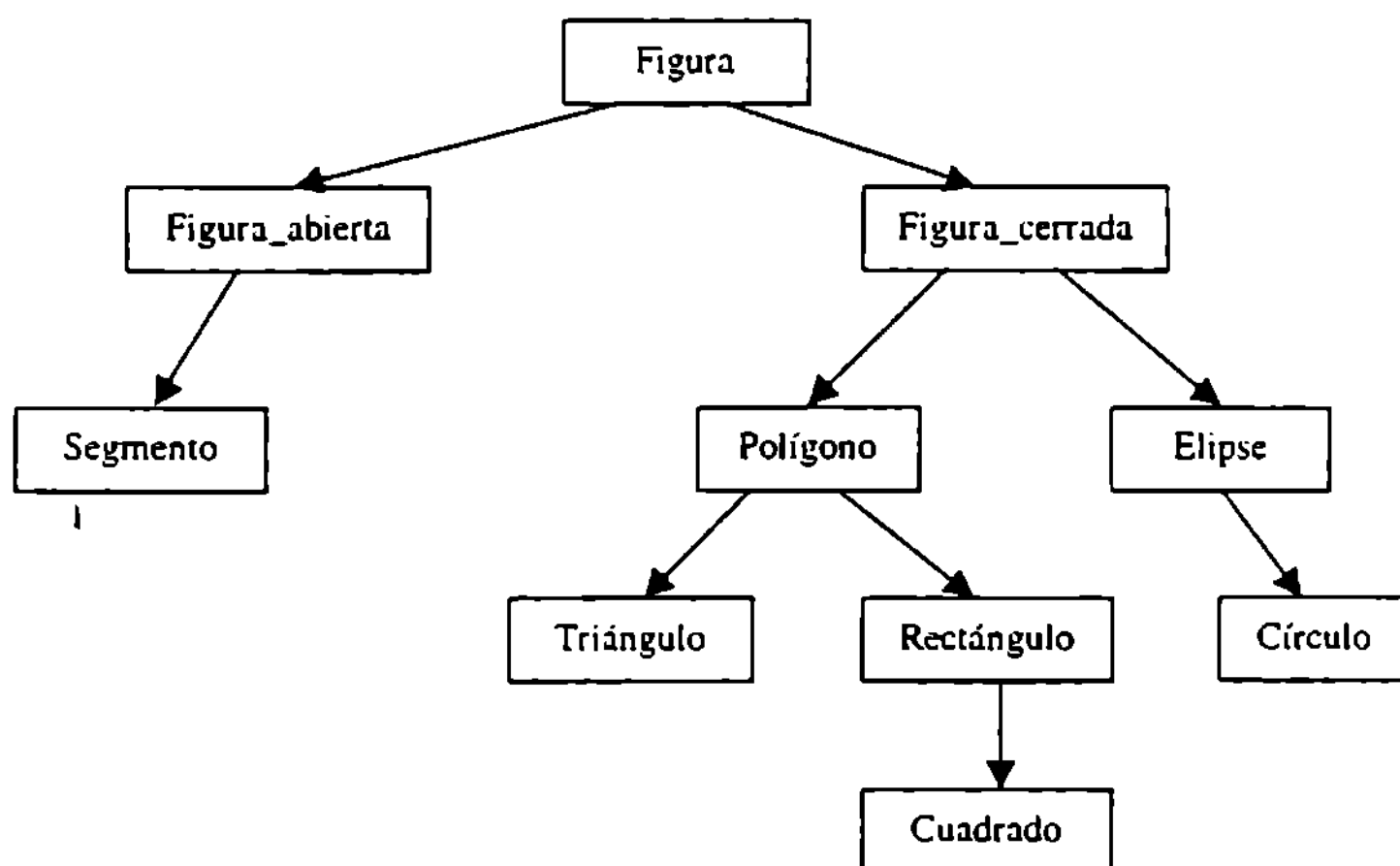
En la figura anterior se puede observar como la clase *Libro* hereda atributos y métodos de la clase *Documento* así como la clase *Libro* deriva en la clase *Novela*.

La herencia puede ser tratada desde dos puntos de vista: Desde el punto de vista del *Módulo* puesto que se extienden las características de una determinada jerarquía de clases sin tener que volver definir los atributos ni codificar los métodos ya definidos. En definitiva estamos añadiendo nueva funcionalidad a un módulo utilizando atributos y métodos ya definidos. Desde el punto de vista del *Tipo* puesto que se pueden instanciar con este mecanismo nuevas instancias de clase. La herencia representa una relación *es\_un* teniendo en cuenta la siguiente *Regla de Asignación* :

La asignación  $X=Y$ , donde  $X$  es una instancia de la clase  $A$  e  $Y$  es una instancia de la clase  $B$ , es válida sólo si la clase  $B$  es descendiente de la clase  $A$ .

#### 14.2.4. Redefinición de Métodos.

Al igual que las clases heredan atributos de sus superclases, también se heredan los métodos definidos. Si se ve la herencia desde el punto de vista del módulo, al derivar una clase se pueden redefinir los métodos heredados para aprovechar las nuevas características que extiende la nueva clase. Imagínese una jerarquía como la que sigue:



Al derivar la clase *Polígono* en la clase *Rectángulo* extendemos la funcionalidad del método *Area*, puesto para un *Rectángulo* se calcula como *base \* altura* mientras que para un *Polígono* habría que utilizar triangulaciones. Dado que la clase *Rectángulo* facilita esta tarea, sería conveniente redefinir el método *Area* que se hereda para que sea calculado de manera más eficiente. Al redefinir un método se anula el método heredado, de manera que cada vez que se invoque al método *Area* se ejecutará el redefinido en la clase *Rectángulo*, no el heredado de la clase *Polígono*.

Cada elemento de la jerarquía puede dar su propia versión de un determinado método: en esto consiste la redefinición.

#### 14.2.5. Ligadura temprana vs ligadura tardía.

Repasando la *Regla de la asignación* expuesta en el punto 14.1.3. se puede observar que si se opera con variables estáticas existe una copia de los atributos de *Y* en los de *X* pero no ocurre lo mismo para los métodos. Luego, si después de esa asignación los únicos métodos que se pueden invocar siguen siendo aquellos que pertenecen a la clase *A* puesto que *X* sigue perteneciendo a dicha clase. La asignación entonces sólo copia parcialmente las características que definen un objeto.

Todo esto es debido a que la variable *X* está definida como variable estática y la referencia a los métodos que se pueden invocar de esa clase se resuelve en tiempo de compilación: *Ligadura Temprana*. Debe existir pues, una forma de resolver estas

referencias a métodos de manera que, una vez realizada la asignación, los métodos que se puedan invocar sean los asociados a la clase *B* y no los asociados a la clase *A*.

Esto se consigue retrasando la resolución de las referencias de los métodos a ejecutar de una clase a tiempo de ejecución: *Ligadura Tardía*. La clave radica entonces en la *vinculación dinámica*. Basta declarar los métodos en conflicto como virtuales y mantener referencias a objetos en vez de variables estáticas.

Los Lenguajes Orientados a Objeto mantienen mecanismos de vinculación dinámica, sin embargo, al conllevar esto un coste adicional, C++ y Pascal, dan la posibilidad al programador de decidir si se desea una resolución dinámica o estática mediante el uso de métodos virtuales.

En definitiva, con esto se consigue tener auténticos objetos polimórficos, al resolverse las referencias de los métodos a ejecutar en tiempo de ejecución, y utilizando vinculaciones dinámicas.

Trabajar con variables dinámicas siempre conlleva un mayor control por parte del programador pero cuando se asigna un puntero a otro no se produce ninguna modificación en las variables referenciadas por éstos.

#### 14.2.6. Generalidad.

Uno de los mayores problemas que mantienen los lenguajes No Orientados a Objetos es la incapacidad de definición de registros parametrizados. Se pueden definir procedimientos y funciones parametrizados, programas parametrizados (aceptan algún tipo de información desde el sistema operativo), módulos "parametrizados", pero no permiten la utilización de estructuras de datos parametrizadas.

Este problema lo siguen teniendo incluso algunos lenguajes de programación orientados a objeto como Borland Pascal: no permiten la definición de *clases* parametrizadas (aunque se puede simular con el uso de vinculación dinámica).

Este mecanismo que se implementa en lenguajes como C++ permiten la utilización de clases de las que se desconoce el *tipo* de sus atributos en tiempo de su definición, para pasarles por parámetro el tipo de los atributos en tiempo de instanciación.

Si en programación tradicional se desea definir dos *Pilas*, una de números enteros y otra de cadena de caracteres, es necesario duplicar los módulos que implementan dichas estructuras con la simple diferencia del nombre y del tipo de datos con el que van a trabajar. Esto supone redundancia de código, lo cual no es aceptable.

A través de la definición de Clases genéricas este problema desaparece al tener un único módulo que implemente las *Pilas* de manera genérica (sin especificar el tipo de datos con el que se trabajará), y especificando a la hora de instanciar ambas pilas el tipo de datos de cada una.

### 14.3. Clases en C++.

Una vez estudiados las principales características de la POO se explicará la implementación de todas estas características en C++, comenzando por cómo definir una *Clase*.

#### 14.3.1. Definición Clases.

Las clases definen el comportamiento de los objetos durante la ejecución del programa, luego tan sólo contienen la especificación de los atributos, *Miembros* en terminología C++, y los prototipos de los Métodos, *Funciones miembro* en terminología C++.

Un primer ejemplo de clase:

```

class punto {
    float x,y;
public:
    punto (float a, b);
    void MoverA (float NuevaX, NuevaY);
    float Distancia (punto otro);
    float CoordX (void);
    float CoordY (void);
}

```

Como se puede observar en el fragmento de código anterior, la definición de una clase implica la utilización nuevas palabras reservadas. De la misma manera que en C se definen estructuras utilizando la palabra clave `struct` en C++ para definir clases se utiliza `class`, a continuación se debe especificar el nombre de la clase.

Entre llaves se especifican los *miembros y funciones miembro* que definen el comportamiento de la clase. Distinguiéndose entre lo que es *privado de clase* y *público* a través de la cláusula `public`. En este caso todo lo que va detrás de dicha cláusula se considera público y todo lo anterior a ella como privado de clase. Se puede especificar además partes privadas de clase usando la cláusula `private` de la misma manera que `public`. Hay que hacer notar que por defecto se toma como activa la cláusula `private`.

Con la definición anterior se definen pues dos miembros de tipo *float* privados de clase y todas las funciones miembro como *publicas*, tal y como indica la metodología de programación orientada a objetos.

Nótese que en la parte *pública* sólo se incluyen los *prototipos* de las funciones miembros, no así su implementación, que se desarrollará como parte externa de la definición de la clase:

```

punto::punto (float a, b) {x=a;y=b;}

void punto::MoverA (float NuevaX, NuevaY)
    {x=NuevaX; y=NuevaY;}

float punto::Distancia(punto otro)
    {return(sqrt( (x-otro.CoordX())**2 + (y - otro.CoordY())**2));}

float punto::CoordX (void) {return(x);}

float punto::CoordY (void) {return(y);}

```

Debido a que la implementación de las funciones miembro son externas a la definición de la clase, en la cabecera se incluye el nombre de la clase a la que pertenece la función miembro seguida de '::'. Dentro de la implementación de la función miembro no se incluye modificación alguna al código C, salvo la llamada a las funciones miembro, que como se observa en el código anterior se especifica *objeto.funcionMiembro()*; como si de una estructura C se tratara.

### 14.3.2. Instanciación de clases: Objetos.

La instanciación de clases consiste en la definición de variables cuyo tipo es el definido por la clase. Como cualquier variable que se define en C las clases se instancian indicando en primer lugar el nombre de la clase seguida de una lista de variables en este caso *Instancias de Clase u Objetos*:

```
punto Origen(0,0);
punto Destino(4.,6.);
```

Con este código se definen dos *Instancias de Clase* punto denominadas *Origen* y *Destino*.

## 14.4. Constructores y Destructores.

### 14.4.1. Constructores.

Un constructor no es más que una *función miembro* que tiene como misión la inicialización de los *miembros* integrantes del objeto y la inicialización de la *Tabla de Métodos Virtuales (VMT)* asociada a la clase.

El constructor de una clase se identifica perfectamente al tener el mismo nombre de la clase. Se trata también de una función miembro que no devuelve tipo, ni siquiera el tipo *void*.

```
class punto {
    float x,y;
public:
    punto (float a, b); // Constructor.
    // resto de funciones miembro.
}
```

Si se trata de una función miembro, ésta debería invocarse a través de la inclusión de una sentencia dentro del código ejecutable de manera explícita. Pero esto no es así en C++.

Existe una peculiaridad en la instanciación de clases que la hace diferente a la instanciación de cualquier otra variable. Como se puede observar en la instanciación de los objetos *Origen* y *Destino* anterior, a la hora de definir un objeto se está invocando de manera automática al constructor asociado a la clase, de manera que no es necesario

incluir una sentencia explícita para invocarlo. El programador queda liberado de esta tarea que a veces es la causa de muchos errores de programación.

#### 14.4.2. Destructores.

Un *Destructor* es otra *función miembro* con la misión específica de la destrucción del enlace a la *VMT* asociada. Además, se debe incluir en la implementación de este tipo de función miembro toda tarea de liberación de memoria dinámica *Heap*, operaciones de cierre de ficheros asociados al objeto, etc. En general reinicialización de los atributos a su estado inicial, de manera que no exista pérdida de funcionalidad del programa.

Así, si al instanciarse una clase se invoca automáticamente al constructor de la clase, se invoca también automáticamente al destructor, en el momento que se alcance el final del ámbito de la instancia. Tampoco es necesaria entonces una llamada explícita al *destructor* por parte del programador.

Dé la misma forma que el *Constructor* de una clase tiene el mismo nombre que la clase, el *Destructor* de una clase se identifica también con el nombre de la clase pero antecedido por el símbolo ~.

```
class punto (
    float x,y;
public:
    punto (float a, b); // Constructor.
    // resto de funciones miembro.
    ~punto(void); // Destructor.
)
```

La implementación de los *constructores* y *destructores* no conlleva ninguna diferencia con respecto al resto de funciones miembro.

#### 14.4.3. Constructores y Destructores por defecto.

C++ define constructores y destructores por defecto. De manera que si no se incluye un constructor en una clase se invocará el definido por defecto. Con esto la siguiente definición de clase sería correcta:

```
class punto (
    float x,y;
public:
    void init (float a, b); // Inicializador.
    // resto de funciones miembro.
)
```

Al no incluirse ningún constructor se invocará al constructor por defecto. Quedaría entonces como responsabilidad del programador el realizar una llamada explícita a la función miembro *init* para inicializar los atributos definidos en la clase:

```
punto origen, destino;
// definición del resto de instancias.
origen.init(1.5, 1.0); destino.init(3.2, 4.7);
```



#### 14.4.4. Argumentos por omisión.

A veces es conveniente especificar los valores por omisión de los argumentos de una función; éstos se utilizan si, al hacer la llamada, se decide no especificar el valor de un argumento. Por ejemplo, el constructor de la anterior clase `punto` es un candidato idóneo para la utilización de esta técnica. El valor predeterminado para el argumento de una función se especifica en la declaración de la misma.

```
class punto {
    float x,y;
public:
    punto (float a=0., b=0.); // Constructor con argumentos por omisión
    // resto de funciones miembro.
    ~punto(void); // Destructor.
}
```

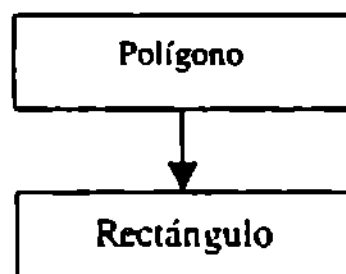
### 14.5. Herencia en C++.

La herencia expresa relaciones entre comportamientos como clasificación, especialización, generalización extensión y aproximación como se estudió con anterioridad.

La herencia en C++ es *múltiple*, *no estricta* y *selectiva*, lo que quiere decir que se puede heredar de más de una clase (*múltiple*), no se impide la redefinición del significado de un método (*no estricta*) y existe la capacidad de decidir qué se hereda y qué no (*selectiva*).

#### 14.5.1. Expresión de la herencia en C++.

Si se pretende expresar en C++ la siguiente jerarquía de clases



Tendremos:

```
class Poligono{
    int num_vertices;
    Punto vertices[MAX];
public:
    Poligono (Punto v[MAX]);
    void Mostrar (void);
    void Ocultar (void);
    float Perimetro(void);
    float Area(void);
    ~Poligono (void);
}
```

```
// definición de métodos de la clase polígono.

class Rectangulo:Poligono{
    // nuevas características de la subclase.
    Public:
        Rectangulo(Punto Origen, float base, altura);
        Float Perimetro (void);
        // nuevas funciones miembros que extiendan la clase.
        -Rectangulo(void);
}
```

Como se puede observar en la definición de la clase *Rectángulo* después del nombre de la clase aparece el nombre de la clase de la que hereda separada por ':'. De esta manera se indica la herencia en C++.

Si se necesitara hacer uso de *herencia múltiple* se especifican las clases de las que se hereda, separándolas por coma.

Como se indica al comienzo del apartado C++ permite distintos tipos de herencia. C++ añade además tres niveles de acceso a los atributos:

- *Público* – `public` – en el que no existen limitaciones en acceso ni para las clases clientes ni para las clases derivadas. Aunque este tipo de acceso está permitido va en contra de la metodología de programación orientada a objetos.
- *Protegido* – `protected` – Se imponen limitaciones a las clases clientes (los miembros son considerados como privados de clase), pero no a las clases derivadas (las funciones miembro de las clases derivadas pueden acceder y modificar los miembros heredados de las clases clientes).
- *Privado* – `private` – Se imponen limitaciones a las clases clientes y a las clases derivadas. En este caso las funciones miembro definidas en las clases derivadas no pueden acceder a los *miembros* heredados de las clases clientes, si no es a través de las *funciones miembro* definidas en las clases clientes.

Con todo esto el fragmento de código anterior podría estar mal en el caso en el que la función miembro *Perimetro* redefinida en la clase *Rectángulo* accediera de manera directa a los miembros definidos en la clase *Poligono*.

La manera de solucionar este problema es la inclusión de la cláusula *protected* a la hora de definir los atributos:

```
class Poligono{
    protected: // añadida para permitir acceso directo a las clases derivadas
        int num_vertices;
        Punto vertices[MAX];
    public:
        Poligono (Punto v[MAX]);
        void Mostrar (void);
        void Ocultar (void);
        float Perimetro(void);
        float Area(void);
        -Poligono (void);
}
```

```
// definición de métodos de la clase polígono.

class Rectangulo:Poligono(
    // nuevas características de la subclase.
public:
    Rectangulo(Punto Origen, float base, altura);
    float Perimetro (void);
    // nuevas funciones miembros
    // que extiendan la clase.
    ~Rectangulo(void);
}
```

## 14.6. Polimorfismo y Sobrecarga de funciones.

### 14.6.1. Polimorfismo.

Por *Polimorfismo* en POO se entiende la capacidad que tienen los objetos para cambiar de forma en tiempo de ejecución. Se expone a continuación como se obtienen objetos polimórficos en C++.

Si se tiene el siguiente fragmento de código:

```
Punto origen (0,0);
Rectangulo r(origen,5,4);
Poligono p;
cout << r.perimetro();
```

Al tratarse *r* de una instancia de la clase *Rectangulo* se invoca a la función miembro *perimetro* redefinida en esa clase.

Pero si después de ese código se incluye el siguiente: --

```
p=r;
cout << p.perimetro();
```

*P* es una variable estática y, a pesar de la asignación, se ejecutará la función miembro *perimetro* de la clase *Poligono*, y no la de *Rectangulo*.

Todo esto ocurre porque la referencia al método perímetro se resuelve en tiempo de compilación (*Ligadura Temprana*).

Para resolver este problema se tiene que utilizar la *vinculación dinámica*. Basta declarar el método *perimetro* virtual y tener referencias a objetos en vez de variables estáticas de la siguiente manera:

```
class Poligono(
    protected: //para permitir acceso directo a las clases derivadas.
        int num_vertices;
        Punto vertices[MAX];
    public:
        Poligono (Punto v[MAX]);
        void Mostrar (void);
        void Ocultar (void);
        virtual float Perimetro(void);
        float Area(void);
        ~Poligono (void);
}
```

```
// definición de métodos de la clase polígono.

class Rectangulo:Poligono{
    // nuevas características de la subclase.
public:
    Rectangulo(Punto Origen, float base, altura);
    Virtual float Perimetro (void);
    // nuevas funciones miembros que extiendan la clase.
    ~Rectangulo(void);
}
```

Con el fragmento anterior se tiene parte del problema solucionado, pero falta incluir el mecanismo de vinculación dinámica como se indica a continuación:

```
Rectangulo *r1;
Poligono *p1;
...
// inserción de los vértices en el polígono.
p1 = new Poligono(vertices);
cout << p1->Perimetro(); // perimetro de clase poligono.
r1 = new Rectangulo(origen,5,4);
}
p1 = r1; // p1 ha cambiado de forma. Es un Rectangulo.
cout << p1->Perimetro(); // Perimetro de clase Rectangulo.
```

De esta manera se consiguen auténticos objetos polimórficos teniendo siempre en cuenta que la *regla de la asignación* se tiene que seguir cumpliendo aunque se trabaje con vinculación dinámica.

#### 14.6.2. Sobrecarga.

Se dice que el nombre de una función está sobrecargado si hay dos o más cuerpos de funciones asociados con el mismo nombre.

Para entenderlo mejor se trata de tener nombres de funciones polimórficos. Los compiladores saben cómo generar código para sumar dos números enteros, así como para sumar dos números reales (*float*). De esta manera tenemos una función *suma* que es capaz de operar con dos tipos de datos diferentes, o dicho de otra manera, la función suma cambia de forma para operar con números enteros y números reales.

Podría parecer que lo que se hace en el caso anterior es redefinir la función suma, pero no es así, puesto que existe un cambio en el tipo de los argumentos (para que exista redefinición las cabeceras de las funciones no pueden cambiar ni en su nombre, ni en el tipo y número de sus argumentos).

Se trata entonces de diferentes implementaciones del código de la función bajo el mismo nombre, en la que cada una de las diferentes versiones tiene un prototipo.

Si el prototipo es el mismo, entonces se tiene *redefinición* y no *sobrecarga*.

La ambigüedad que aparece ante la pregunta ¿Qué código se ejecuta de los diferentes que se tiene? se evita mediante *signaturas*, compuestas por el nombre de la función, número y tipo de sus argumentos.

¿Cómo distinguir Sobrecarga de Redefinición?

- **Sobrecarga:** Funciones miembro con el mismo nombre pero con comportamientos en los que no existe relación alguna. Sobrecarga no conlleva anulación del resto de funciones miembro implicadas.
- **Redefinición:** Las diferentes versiones poseen la misma semántica y mismo nombre pero con implementaciones distintas. Redefinición conlleva anulación del resto de funciones miembro implicadas.

Un ejemplo de Sobrecarga:

```
class complejo{
    float r,i;
public:
    complejo (float x=0., float y=0.);
    float ParteReal(void);
    float ParteImaginaria(void);
    complejo operator + (complejo);
    int operator == (complejo);
    complejo operator = (complejo);
}

complejo::complejo (float x, float y)
{ r=x; i=y;}

float complejo::ParteReal(void) {return (r);}

float complejo::ParteImaginaria (void) {return(i);}
complejo complejo::operator + (complejo a)
{
    complejo temp;
    temp.r = r+a.r;
    temp.i = i+a.i;
    return(temp);
}

int complejo::operator==(complejo a)
{return( (a.ParteReal() == r) &&
        (a.ParteImaginaria() == i)?1:0);}

complejo complejo::operator = (complejo a)
{
    this->r=a.r;
    this->i=a.i;
    return(*this);
}

void main()
{
    complejo a,b,CERO;
    complejo C(1.,1.);
    if(!CERO==c) c=a+b;
}
```

## 14.7. Uso de Templates.

Un *Template* no es más que la denominación que en C++ se da a las clases genéricas mencionadas con anterioridad en este capítulo. Se trata entonces de definir clases parametrizadas de manera que el tipo de datos que van a contener no se especifique hasta la instanciación de las clases.

A continuación se muestra un ejemplo de clase genérica implementada en C++:

```
template <class Tipo>
class pila{
    Tipo *datos;
    int top, num;
public:
    pila (int max = 10);
    void insertar (Tipo c);
    Tipo *cima(void);
    int vacia(void);
    void borrar(void);
    ~pila();
};

template <class Tipo>
pila<Tipo>::pila(int max)
{
    datos = new Tipo(max);
    top = -1;
    num = max;
}

template <class Tipo>
pila<Tipo>::~~pila()
{
    delete [] datos;
}

//Restantes funciones miembro.
```

Una vez definida con un tipo de datos genérico *Tipo* es en tiempo de instanciación cuando se le da valor al tipo de datos genérico:

```
pila<char> pc;
pila<int> pi(30);
pila <char *> pcad(20);
```

En el ejemplo se definen tres pilas: pc es una pila de caracteres, pi es una pila de 30 enteros y pcad es una pila de 20 punteros a carácter (cadenas).

## *Un ejemplo de programa en C++*

---

### **15.1. Introducción.-**

En este capítulo se presenta un ejemplo completo de programa en C++. Se trata del mismo ejemplo expuesto en el capítulo 12 pero utilizando la metodología de programación orientada a objetos. Con este ejemplo se pretende mostrar el uso de la mayoría de características que el lenguaje de Programación C++ aporta, para su posterior comparación con C.

## 15.2. El Código.-

```

#include <stdio.h>

/*****
/* Programa Agenda */
*****/
/* Fecha : 1 de Octubre de 1997 */
/* Autor: Juan Manuel Murillo Rodríguez */
/* Versión: 1.0 */
/* Comentario: Versión inicial del programa */
/* Agenda. */
*****/
/* Fecha : 10 de Diciembre de 1997 */
/* Autor: Jorge Quirós Rosado */
/* Versión: 2.0 */
/* Comentario: Implementación del sistema */
/* en C++. */
*****/

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>

/***** ESTRUCTURAS DE DATOS *****/

/* Información a guardar en la agenda */
typedef struct datos_agenda
{
    char fecha[9];          /* Fecha de la anotación en la agenda */
    char hora[6];          /* Hora de la anotación en la agenda */
    char texto[160];        /* Texto de la anotación */
} datos_anotacion;

/* Definición de la clase genérica Lista */
template <class Tipo>
class lista {

    /* Definición de un Nodo de la lista */
    typedef struct nodo_lista_d_e{
        Tipo anotacion;
        struct nodo_lista_d_e *sig;
        struct nodo_lista_d_e *ant;
    } nodo_lista;

    /* La lista se compone de tres elementos */
    nodo_lista *cabeza;    /* Puntero al siguiente */
    nodo_lista *cola;      /* Puntero al anterior */
    nodo_lista *actual;    /* Texto de la anotación */

public:
    /***** OPERACIONES DEL TAD LISTA *****/
    lista (Tipo anot); // Constructor.
    int vacia (void);
    int final (void);
    int principio (void);
    int primer_nodo (void);
    int insertar (Tipo anot);
    void eliminar (void);
    void avanzar (void);
    void retroceder (void);
    void extraer (Tipo &anot);
    void ir_principio (void);
    void ir_primer_nodo (void);
    void ir_final (void);
};

/***** OPERACIONES DEL PROGRAMA *****/

void init_datos_anotacion (datos_anotacion &aux);
void mostrar_datos_anotacion (datos_anotacion aux);
char imprimir_pantalla (lista<datos_anotacion> &l);

```



```

lista<datos_annotacion> ejecutar_opcion (lista<datos_annotacion> &l, char op);
lista<datos_annotacion> opcion_insertar_ord (lista<datos_annotacion> &l);
lista<datos_annotacion> insertar_ord (lista<datos_annotacion> &l,
                                     datos_annotacion nueva_annot);
lista<datos_annotacion> opcion_modificar (lista<datos_annotacion> &l);

void opcion_cargar      (lista<datos_annotacion> &l);
void opcion_salvar      (lista<datos_annotacion> &l);
void convertir_fecha    (char *fecha, char *fecha_sal);

/*===== IMPLEMENTACION DE LAS FUNCIONES DEL TAD =====*/

template <class Tipo>
lista<Tipo>::lista (Tipo anot)

/* CONSTRUCTOR: Crea un lista doble enlazada. Inicialmente se crea la lista
con un registro bandera. Esto simplifica los algoritmos de inserción y
borrado. El nodo bandera siempre quedará en la cabeza de la lista ya que
las inserciones se realizarán siempre delante del nodo actual */

{
    nodo_lista *nuevo_nodo;

    nuevo_nodo = (nodo_lista *) malloc(sizeof(nodo_lista));
    nuevo_nodo -> sig = NULL;
    nuevo_nodo -> ant = NULL;
    nuevo_nodo -> anotacion = anot;
    cabeza = nuevo_nodo;
    cola = nuevo_nodo;
    actual = nuevo_nodo;
}

template <class Tipo>
int lista<Tipo>::vacía (void)

/* Devuelve true cuando la lista est vacía */

{ return (cabeza == cola); }

template <class Tipo>
int lista<Tipo>::final (void)

/* Devuelve true cuando el nodo actual de la lista es el último elemento de
ésta. */

{ return (actual == cola); }

template <class Tipo>
int lista<Tipo>::principio (void)

/* Devuelve true cuando el nodo actual es el nodo bandera situado en la
cabeza de la lista */

{ return (actual == cabeza); }

template <class Tipo>
int lista<Tipo>::primer_nodo (void)

/* Devuelve true cuando el nodo actual es el primer nodo de la lista.
Después del nodo bandera. */

{ return (actual == cabeza->sig); }

```

```

template <class Tipo>
int lista<Tipo>::insertar (Tipo anot)

/* Inserta un nuevo elemento en la lista. Las inserciones se realizarán
siempre delante del nodo actual. Si no se dispusiera de un nodo bandera en
el inicio de la lista, se tendría que tener una función específica para
realizar inserciones delante del primer nodo de la lista. */

{
    nodo_lista *nuevo_nodo;

    /* Tomamos espacio para un nuevo nodo */
    if ((nuevo_nodo = (nodo_lista *) malloc(sizeof(nodo_lista))) == NULL)
    {
        cout << "No ha memoria suficiente para la inserción \n";
        return 0;
    }
    else
    {
        nuevo_nodo->anotacion = anot;
        nuevo_nodo->sig = actual->sig;
        nuevo_nodo->ant = actual;
        if (! final()) /* Si se está al final no tengo nodo siguiente */
            nuevo_nodo->sig->ant = nuevo_nodo;
        else /* Si se ha insertado al final se tiene que */
            cola = nuevo_nodo; /* modificar la cola de la lista. */
        actual->sig = nuevo_nodo;
        actual = nuevo_nodo;
        return 1;
    }
}

template <class Tipo>
void lista<Tipo>::eliminar (void)

/* Elimina el nodo actual de la lista. El nodo actual pasa a ser el
siguiente del nodo que se ha eliminado a menos que fuese la cola, en cuyo
caso, el próximo nodo actual será el antecesor del que se acaba de eliminar
*/

{
    nodo_lista *nodo;

    if (! vacia()) /* Si la lista esta vacia no se puede eliminar */
    {
        nodo = actual;
        actual->ant->sig = actual->sig;
        if (! final())
        {
            actual->sig->ant = actual->ant;
            actual = actual->sig;
        }
        else
        {
            actual = actual->ant;
            cola = actual;
        }
        nodo->sig = NULL; /* Se libera el espacio ocupado por */
        nodo->ant = NULL; /* el nodo que se acaba de eliminar */
        delete nodo;
    }
}

template <class Tipo>
void lista<Tipo>::avanzar (void)

/* Hace que el nodo actual pase a ser el siguiente del presente nodo actual.
Si el nodo actual es la cola de la lista, no se hace nada. */

{
    if (! final())
        actual = actual->sig;
}

```

```

template <class Tipo>
void lista<Tipo>::retroceder (void)

/* Hace que el nodo actual pase a ser el antecesor del presente nodo actual.
Si el nodo actual es el nodo bandera, es decir, si nos encontramos
al principio de la lista, no se hace nada. */

(
    if (! principio ())
        actual = actual->ant;
)

template <class Tipo>
void lista<Tipo>::extraer (Tipo &anot)

/* Devuelve los datos contenidos en el nodo actual. */

(
    anot = actual->anotacion; // se devuelve a través de una referencia.
)

template <class Tipo>
void lista<Tipo>::ir_principio (void)

/* Hace que el nodo actual pase a ser el nodo bandera situado al comienzo de
la lista */

(
    if (! vacia ())
        actual = cabeza;
)

template <class Tipo>
void lista<Tipo>::ir_primer_nodo (void)

/* Hace que el nodo actual pase a ser el primer nodo de la lista, es decir,
el siguiente al nodo bandera. */

(
    if (! vacia ())
        actual = cabeza->sig;
)

—

template <class Tipo>
void lista<Tipo>::ir_final (void)

/* Hace que el nodo actual pase a ser la cola de la lista */

(
    if (! vacia ())
        actual = cola;
)

/*===== PROGRAMA PRINCIPAL =====*/

void main (void)
(
    datos_anotacion bandera = ("S", "S", "S");
    lista<datos_anotacion> l(bandera);
    char opcion;

    do
    (
        opcion = imprimir_pantalla (l);
        l = ejecutar_opcion (l,opcion);
    )
    while (!(opcion == 'X') && !(opcion == 'x'));
)

```

```

void init_datos_annotacion (datos_annotacion &aux)

/* Función que inicializa los datos de una estructura de tipo
datos_annotacion */

{
    strcpy(aux.fecha, " ");
    strcpy(aux.hora, " ");
    strcpy(aux.texto, " ");
}

void mostrar_datos_annotacion (datos_annotacion aux)

/* Función que muestra los datos de una estructura de tipo datos_annotacion
*/

{
    cout << "   Fecha   : " << aux.fecha << "\n";
    cout << "   Hora    : " << aux.hora << "\n";
    cout << "   Texto   : " << aux.texto << "\n\n";
}

1 char imprimir_pantalla (lista<datos_annotacion> &l)

/* Muestra, en caso de que la lista no esté vacía, el nodo actual, su
antecesor (si existe) y su sucesor (si existe). Además, muestra el menú de
las diferentes operaciones que se pueden realizar sobre la lista. Devuelve
la opción elegida por el usuario. */

{
    datos_annotacion aux;
    char opcion;

    init_datos_annotacion(aux);
    if (! (l.principio() || l.primer_nodo()))
    {
        l.retroceder();
        l.extraer(aux);
        l.avanzar();
    }
    clrscr();
    cout << "                                           VISUALIZACION AGENDA\n";
    cout << "REGISTRO ANTERIOR :\n";
    mostrar_datos_annotacion (aux);
    init_datos_annotacion (aux);
    if (! l.vacia())
        l.extraer(aux);
    cout << "REGISTRO ACTUAL :\n";
    mostrar_datos_annotacion (aux);
    init_datos_annotacion (aux);
    if (! l.final())
    {
        l.avanzar();
        l.extraer(aux);
        l.retroceder();
    }
    cout << "REGISTRO SIGUIENTE :\n";
    mostrar_datos_annotacion (aux);
    cout << "\n\nOPCIONES :\n";
    cout << "   A Avanzar           M Modificar\n";
    cout << "   R Retroceder       S Salvar\n";
    cout << "   I Insertar         C Cargar\n";
    cout << "   E Eliminar         X Salir           Opcion : ";
    opcion = getch();
    return (opcion);
}

```

```

lista<datos_annotacion> ejecutar_opcion (lista<datos_annotacion> &l, char op)

/* En función de la opción elegida por el usuario determina que función o
procedimiento ha de ejecutarse. Devuelve la misma lista que recibió como
parámetro con las modificaciones que haya sufrido tras la ejecución de la
operación. */

{
    if ((op=='a') || (op=='A'))
        l.avanzar();
    else
        if ((op=='r') || (op=='R'))
            l.retroceder();
        else
            if ((op=='i') || (op=='I'))
                l = opcion_insertar_ord (l);
            else
                if ((op=='e') || (op=='E'))
                    l.eliminar();
                else
                    if ((op=='m') || (op=='M'))
                        l = opcion_modificar (l);
                    else
                        if ((op=='s') || (op=='S'))
                            opcion_salvar (l);
                        else
                            if ((op=='c') || (op=='C'))
                                opcion_cargar (l);
                            return l;
    }
}

```

```

lista<datos_annotacion> opcion_insertar_ord (lista<datos_annotacion> &l)

```

/\* Inserta un nuevo nodo en la lista utilizando para ello la función insertar\_ord. Antes de insertar el nodo pide al usuario que introduzca los datos de la nueva anotación. \*/

```

{
    datos_annotacion nueva_annot;

    clrscr();
    cout << "INSERCIÓN ORDENADA DE UNA NUEVA ANOTACIÓN.\n\n\n";
    cout << "Introduzca los datos de la anotación :\n\n";
    cout << "    Fecha de la anotación (DD/MM/AA) : ";
    cin  >> nueva_annot.fecha;
    cout << "    Hora de la anotación (HH:MM)      : ";
    cin  >> nueva_annot.hora;
    cout << "    Texto de la anotación (Max. 160) : ";
    cin  >> nueva_annot.texto;
    l = insertar_ord (l,nueva_annot);
    return l;
}

```

```

lista<datos_annotacion> insertar_ord (lista<datos_annotacion> &l,
                                     datos_annotacion nueva_annot)

```

/\* Inserta un nuevo nodo en la lista. La inserción se hará de forma ordenada atendiendo a la fecha de la nueva anotación. Puesto que todas las inserciones se harán con esta función conseguiremos una lista donde todos los nodos estarán ordenados por fecha y hora. Lo primero que realiza la función es pedir al usuario que introduzca los datos de la anotación a insertar. \*/

```

{
    datos_annotacion actual;
    char factual[7];
    char finserter[7];
    char hinsertar[7];
    char hactual[7];
    long lfinserter,lfactual,lhinsertar,lhactual;

    convertir_fecha (nueva_annot.fecha,finserter);
    finserter[2] = '0';
    lfinserter = atol(finsterar);
    strcpy(hinsertar,nueva_annot.hora);
    hinsertar[2] = '0';
    lhinsertar = atol(hinsterar);
}

```

```

    if (l.vacia())
        l.insertar(nueva_anot);
    else
    {
        l.ir_principio();
        do
        {
            /* Buscamos el lugar correcto para */
            /* la insercion ordenada */
            l.avanzar();
            l.extraer (actual);
            convertir_fecha(actual.fecha, factual);
            lfactual = atol(factual);
            strcpy(hactual, actual.hora);
            hactual[2] = '0';
            lhactual = atol(hactual);
        }
        while (((lfinsertar > lfactual) ||
            ((lfinsertar == lfactual) && (lhinsertar > lhactual))) &&
            (! l.final ()));
        if ((lfinsertar < lfactual) ||
            ((lfinsertar == lfactual) && (lhinsertar < lhactual)))
            l.retroceder ();
        l.insertar (nueva_anot);
    }
    return l;
}
}

```

```

lista<datos_annotacion> opcion_modificar (lista<datos_annotacion> &l)

```

/\* Esta opción permite modificar los datos pertenecientes a una anotación y que est n contenidos en un nodo. Se permiten modificar todos los datos de la anotación. Esta función hace uso de funciones de comparación de strings. \*/

```

{
    datos_annotacion nueva_anot, actual;

    clrscr();
    l.extraer(actual);
    l.eliminar();
    strcpy(nueva_anot.fecha, "");
    strcpy(nueva_anot.hora, "");
    strcpy(nueva_anot.texto, "");
    cout << "MODIFICACION DE LOS DATOS DE LA ANOTACION ACTUAL.\n\n\n";
    cout << "Introduzca los datos modificados de la anotacion :\n\n\n";
    cout << "Fecha de la anotacion (DD/MM/AA): [" << actual.fecha << "]" ";
    cin >> nueva_anot.fecha;
    cout << "Hora de la anotacion (HH:MM)      : [" << actual.hora << "]" ";
    cin >> nueva_anot.hora;
    cout << "Texto de la anotacion (Max. 150): [" << actual.texto << "]" ";
    cin >> nueva_anot.texto;
    if (0 == strcmp(nueva_anot.fecha, ""))
        strcpy (nueva_anot.fecha, actual.fecha);
    if (0 == strcmp(nueva_anot.hora, ""))
        strcpy (nueva_anot.hora, actual.hora);
    if (0 == strcmp(nueva_anot.texto, ""))
        strcpy (nueva_anot.texto, actual.texto);
    l = insertar_ord (l, nueva_anot);
    return l;
}

```

```

void opcion_cargar (lista<datos_annotacion> &l)

```

/\* Pide al usuario que introduzca un nombre de archivo, lo abre y lee sus registros insertándolos a continuación en la lista. Para ello en principio vacía la lista. El archivo ha de contener informacion sobre anotaciones ordenadas por fecha y hora puesto que de lo contrario el funcionamiento del programa no sería correcto. \*/

```

{
    char fichero[13];
    FILE *f;
    datos_annotacion *anot;

    clrscr();
    cout << "Introduzca el nombre del que se quiere cargar : ";
    cin >> fichero;
    if ((f = fopen(fichero, "rb")) == NULL)
    {

```

```

        char aux;
        cout << "El archivo no pudo abrirse. Pulse return para continuar.";
        aux = getch();
    }
    else
    {
        while (!l.vacia())
            l.eliminar();
        fread(anot, sizeof(datos_annotacion), 1, f);
        while (!feof(f))
        {
            l.insertar(*anot);
            fread(anot, sizeof(datos_annotacion), 1, f);
        }
        l.ir_primer_nodo();
        fclose(f);
    }
}

void opcion_salvar (lista<datos_annotacion> &l)
/* Salva los datos de las anotaciones de la lista actual en un archivo. Para
ello, inicialmente se pide al usuario que introduzca el nombre del archivo en
el que desea volcar la información. */
{
    char fichero[13];
    FILE *f;
    datos_annotacion anot;

    clrscr();
    if (!l.vacia())
    {
        cout << "Introduzca el nombre del archivo sobre el que salvar: ";
        cin >> fichero;
        if ((f = fopen(fichero, "wb")) == NULL)
        {
            char aux;
            cout << "El archivo no pudo abrirse. Pulse una tecla.";
            aux = getch();
        }
    }
    else
    {
        l.ir_primer_nodo();
        do
        {
            l.extraer(anot);
            fwrite(&anot, sizeof(datos_annotacion), 1, f);
            l.avanzar();
        }
        while (!l.final());
        l.extraer(anot);
        fwrite(&anot, sizeof(datos_annotacion), 1, f);
        fclose(f);
    }
}

void convertir_fecha (char *fecha, char *fecha_sal)
{
    char aux[7];

    aux[6] = fecha[8];
    aux[0] = fecha[6];
    aux[1] = fecha[7];
    aux[2] = fecha[3];
    aux[3] = fecha[4];
    aux[4] = fecha[0];
    aux[5] = fecha[1];
    strcpy (fecha_sal, aux);
}

```





## *Modelos de memoria*

Los compiladores de C diseñados para trabajar sobre ordenadores tipo PC, gestionan la memoria dividiéndola en varios bloques de trabajo. De esta forma, hay una zona de la memoria donde se guarda el código, otra para los datos, otra para la pila del sistema, etc.

Debido a los problemas de paginación de memoria de los microprocesadores de la familia *INTEL 80x86* (o compatibles), en los que la memoria se encuentra dividida en bloques de 64 Kb (*segmentos*), de manera que el direccionamiento de la memoria se hace por segmentos y desplazamientos dentro de cada segmento, el tamaño de estas zonas de memoria que manejan los compiladores se ve notablemente influido por el tamaño de los segmentos.

Lo ideal sería que toda la memoria que maneje el compilador pudiera estar dentro de un único segmento. En este caso, las direcciones de memoria se reducirían a un desplazamiento (*offset*), ya que el segmento siempre sería el mismo. Pero en la mayoría de los casos, los requerimientos de memoria de un programa C superan los 64 Kb de un segmento.

Los compiladores de C permiten determinar el tamaño de cada una de las zonas de memoria con las que se va a trabajar, de forma que seleccionemos el tamaño en función de los requerimientos de nuestro programa. Para ello, disponemos de 6 modelos de memoria, cada uno de los cuales destina cierta cantidad de memoria para cada zona.

Estos modelos de memoria, ordenados de menor a mayor complejidad, se muestran en la figura A.1 :

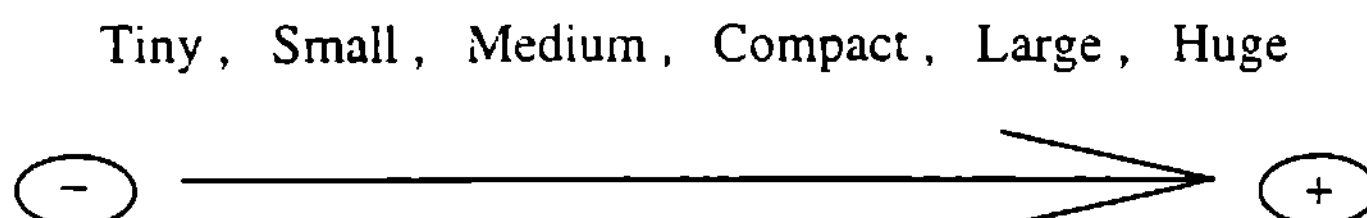
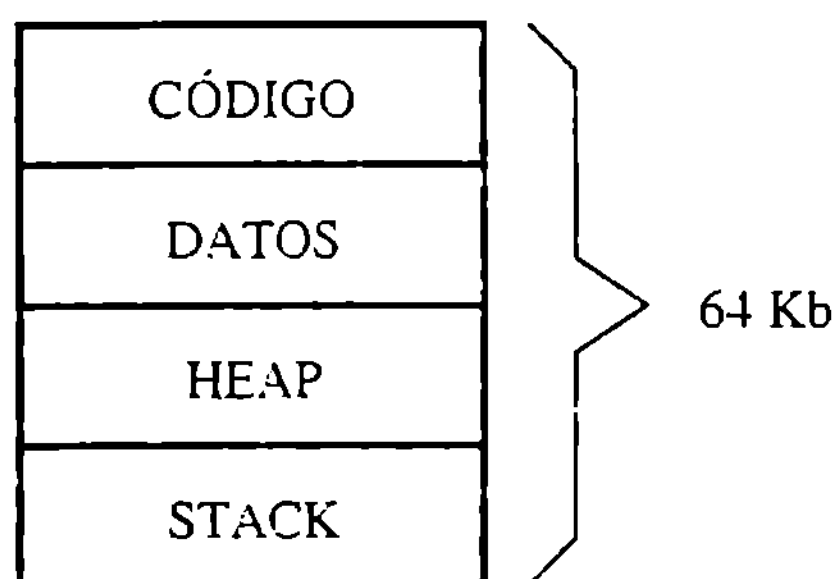


Figura A.1. Modelos de Memoria.

I

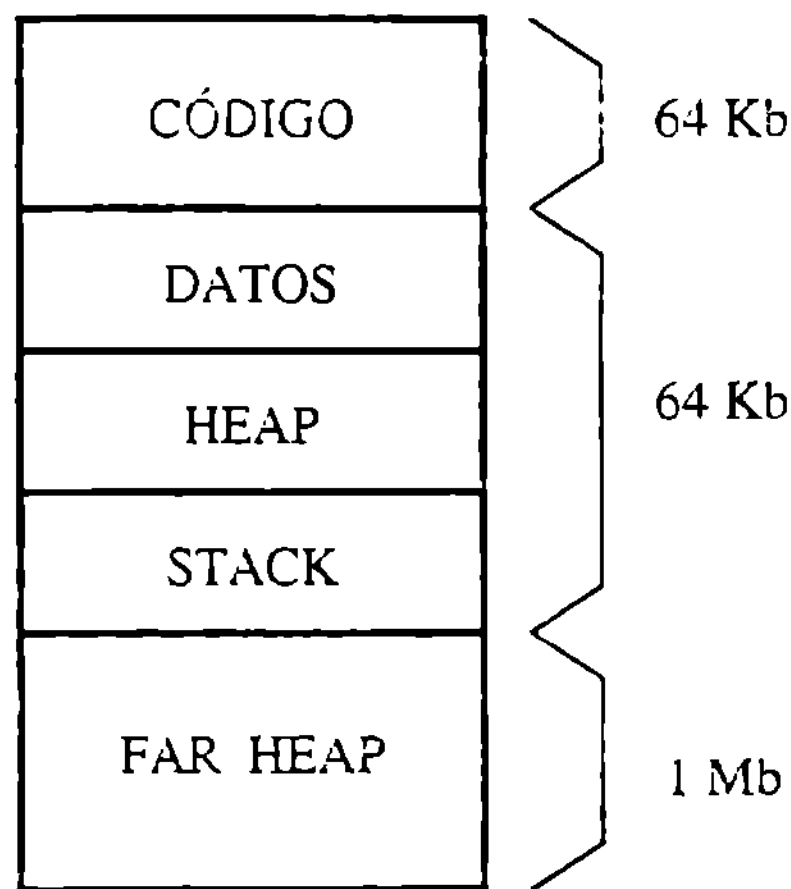
A medida que los modelos se hacen más complejos, trabajan con más memoria y por tanto se hacen más lentos. Por esta razón, se debe usar siempre el modelos más pequeño para el que funcione el programa. Vamos a ir viendo cada uno de estos modelos y los tamaños de las zonas de memoria que cada uno dedica.

### Modelo Tiny



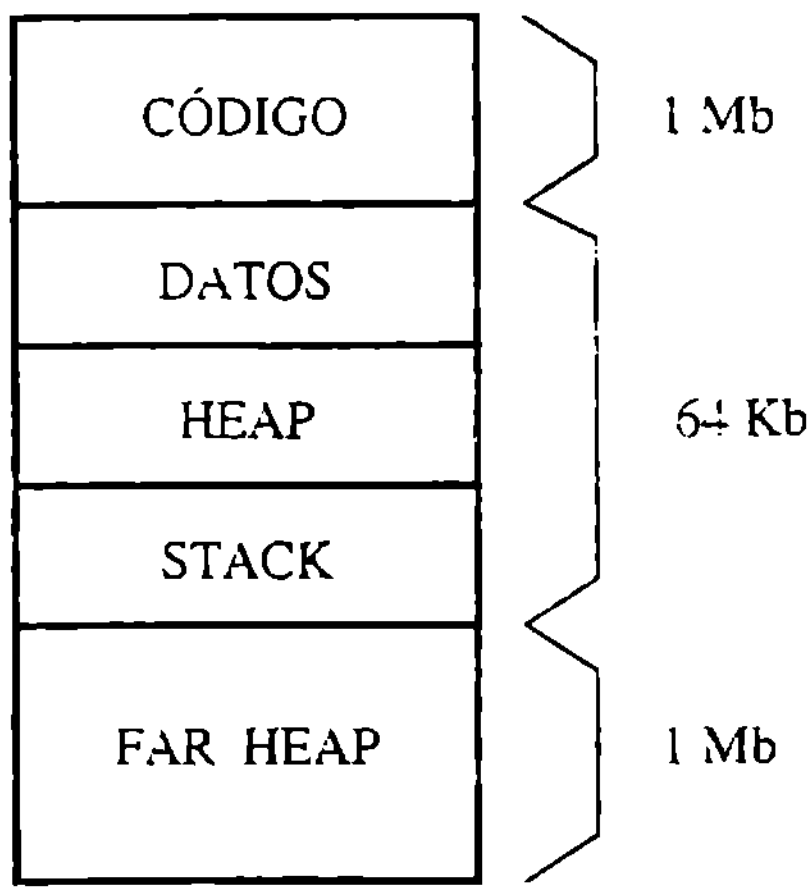
Es el modelo más pequeño, se trabaja con un solo segmento, en el que deben caber todos los datos del programa. Es el más rápido porque las direcciones son desplazamientos (*offset*). Se utilizaba en los ficheros (.COM). Normalmente, cualquier programa cuya versión ejecutable (.EXE) ocupe más de 20 Kb, debe compilarse con un modelo mayor que el modelo Tiny.

Modelo Small



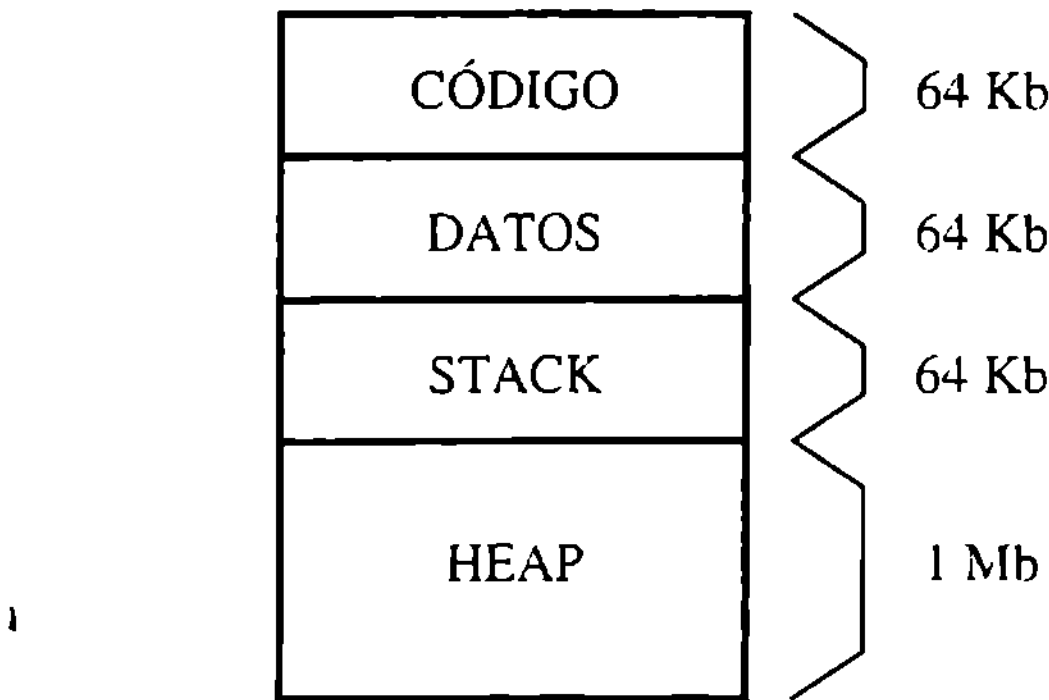
No es recomendable hacer uso de instrucciones *“far heap”* con este modelo, a pesar de que se supone que las soporta. Para ello usar mejor el siguiente modelo *medium*. Es el modelo de memoria que se suele utilizar para programas “normalitos”.

Modelo Medium



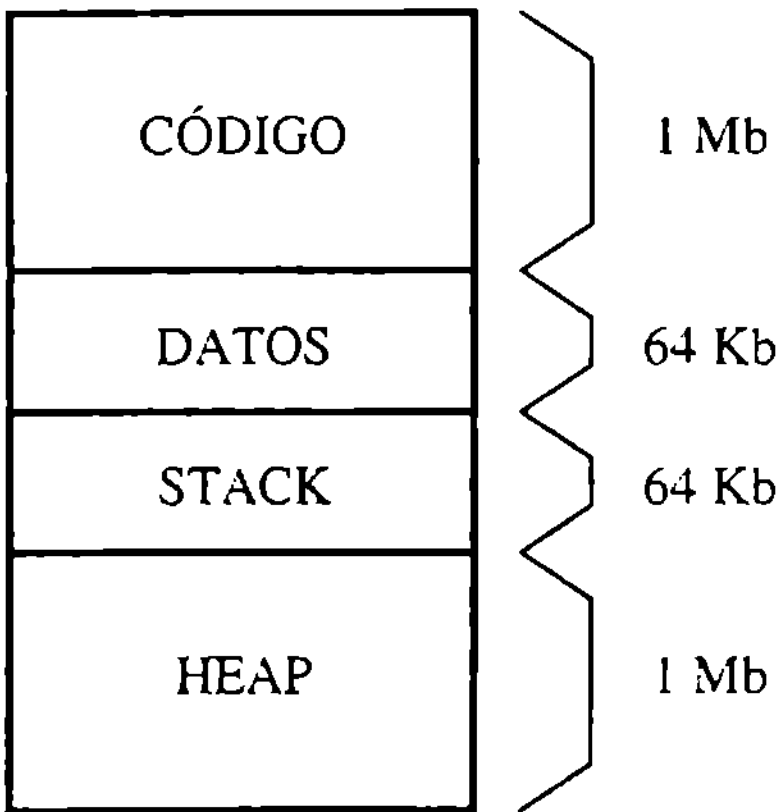
Es el modelo de memoria más adecuado para usar con programas con mucho código y poco uso de memoria. Apto para usar instrucciones “*far heap*”.

**Modelo Compact**



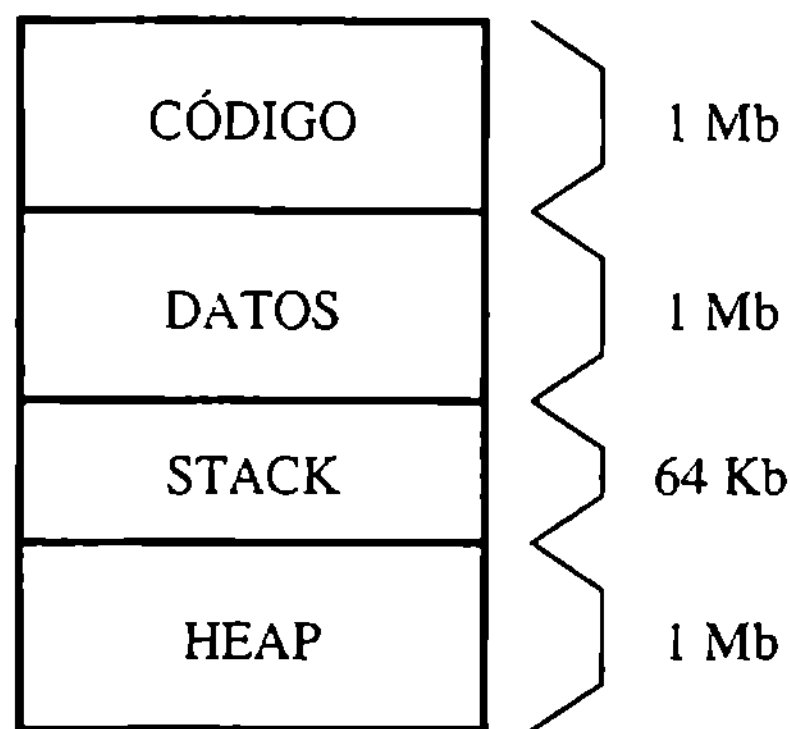
Este modelo es ideal para programas con relativamente poco código y mucho uso de memoria. El tamaño de pila (stack) pasa a ser de 64Kb, el máximo posible.

**Modelo Large**



Para programas muy grandes. Se hace mucho más lento...

## Modelo Huga



Es el modelo más lento. Pero para determinados programas es imprescindible. Es el único modelo de memoria que utiliza **normalización de punteros**.

Internamente, las direcciones de memoria compuestas por un segmento y un desplazamiento se transforman a una sola componente, de la siguiente forma :

- 0000:0120 se transforma en : 0000  
0120  
-----  
00120

Pero puede ocurrir que varias direcciones distintas, después de la transformación, sean la misma dirección :

**0000:0120 se transforma en :      0000**

**0120**

-----

**00120**

```
0010:0020    se transforma en :      0010  
                                           0020  
                                           -----  
                                           00120
```

```
0012:0000    se transforma en :      0012
                                                0000
                                                -----
                                                00120
```

Tres direcciones diferentes dan lugar a la misma dirección final. El modelo de memoria **huge** evita este problema, normalizando los punteros antes de hacer cualquier acceso a memoria, de ahí que ralentice mucho todas las operaciones.

Siempre que haya operaciones de comparación de punteros se debe emplear el modelo **huge** para que dichas comparaciones sean correctas. El proceso de normalización consiste en realizar la transformación y luego separar el resultado en dos partes, empleando luego direcciones de 32 bits. Por ejemplo :

2F84:0532	se transforma en :	2F84
		0532
		-----
		<b>2FD72</b>

¡ y finalmente se separa en :    **2FD7:0002**

## *Librerías Estándar del C*

### **Las librerías estándar del C.-**

Las librerías de C (también llamadas bibliotecas) contienen el código objeto de las funciones proporcionadas con el compilador.

Aunque las librerías (ficheros con extensión .LIB) son parecidas a los ficheros objetos (fichero con extensión .OBJ), existe una diferencia crucial. No todo el código de una librería se añade al programa. Cuando se enlaza un programa que consta de varios ficheros objetos, todo el código de cada fichero objeto se convierte en parte del programa ejecutable final. Esto ocurre se esté o no utilizando el código. En otras palabras, todos los ficheros objetos especificados en tiempo de enlace se unen para formar el programa. Sin embargo, este no es el caso de los ficheros de librería.

Una librería es una colección de funciones. A diferencia de un fichero objeto, un fichero de librería guarda una serie de información para cada función de tal forma que cuando un programa hace referencia a una función contenida en una librería, el enlazador toma esta función y añade el código objeto al programa. De esta forma sólo se añaden al fichero ejecutable aquellas funciones que realmente se utilicen en el programa.

Para utilizar una función de librería debemos incluir su correspondiente fichero de cabecera, para que nuestro programa conozca el prototipo de la función a utilizar. En los ficheros de cabecera (suelen tener extensión .H) además de los prototipos de las funciones puede haber más información, como macros, declaración de tipos, declaración de variables globales, etc.

Los ficheros definidos por el estándar ANSI se presentan en la siguiente tabla.

Fichero	Propósito
assert.h	Define la macro <code>assert()</code> .
ctype.h	Uso de caracteres.
float.h	Define valores en coma flotante dependiente de la implementación.
limits.h	Define los límites dependientes de la implementación.
locale.h	Soporta la función <code>setlocale()</code> .
math.h	Definiciones utilizadas por la librería matemática.
setjmp.h	Soporta saltos no locales.
signal.h	Define los valores de señal.
stdarg.h	Soporta listas de argumentos de longitud variable.
stddef.h	Define algunas constantes de uso común.
stdio.h	Soporta la E/S de fichero.
stdlib.h	Otras declaraciones.
string.h	Soporta funciones de cadena.
time.h	Soporta las funciones de tiempo del sistema.

**Turbo C** añade los siguientes ficheros :

Fichero	Propósito
alloc.h	Asignación dinámica.
bios.h	Funciones relacionadas con la ROM BIOS.
conio.h	E/S por consola.
dir.h	Directorio.
dos.h	Sistema operativo DOS.
errno.h	Errores del sistema.
fcntl.h	Constantes simbólicas utilizadas por <code>open()</code> .
graphics.h	Gráficos.
io.h	E/S estándar a bajo nivel.
mem.h	Funciones de memoria.
process.h	Funciones de proceso.
share.h	Constantes simbólicas utilizadas por <code>sopen()</code> .
sys\stat	Información de ficheros.
sys\timeb	Hora actual.
sys\types	Definición de tipos.
values.h	Constantes simbólicas para compatibilidad con UNIX.



OBSERVACIONES:

Aquellas funciones que tienen al principio de la descripción: (TC), no están en el ANSI C y sí en TURBO C.

A continuación se detallan algunas de las principales librerías del TURBO C. Este apéndice no pretende ser un completo manual de librerías, sino servir al usuario como guía de consultas, donde pueda encontrar un referencia de las funciones que se hallan en las librerías reseñadas.

Por esta razón, hemos elegido las 10 librerías que consideramos más usuales. Dentro de cada una de ellas, se describen los prototipos de las funciones más importantes, así como un breve comentario a cerca de su utilidad.

Por seguir un cierto orden, en primer lugar aparecen las librerías del ANSI C, y a continuación las que añade TURBO C. Dentro de cada librería aparecen por orden alfabético.

CTYPE.H

En esta librería se definen macros relacionadas con el manejo de caracteres.

Macro	Verdad (true) si c es...
isalnum (c)	una letra o dígito
isalpha (c)	una letra
isdigit (c)	un dígito
isctrl (c)	un carácter de borrado o un carácter de control ordinario
isascii (c)	un carácter ASCII válido
isprint (c)	un carácter imprimible
isgraph (c)	un carácter imprimible, excepto el carácter espacio
islower (c)	una letra minúscula
isupper (c)	una letra mayúscula
ispunct (c)	un carácter de puntuación
isspace (c)	un espacio, tabulador, retorno de carro, nueva línea, tabulación vertical, o alimentación de línea
isxdigit (c)	un dígito hexadecimal

Además, Turbo C añade :

Macro	Verdad (true) si c es...
<code>_toupper(c)</code>	En el rango [a-z] a caracteres [A-Z]
<code>_tolower(c)</code>	En el rango [A-Z] a caracteres [a-z]
<code>toascii(c)</code>	Mayor de 127 al rango 0-127 poniendo todos los bits a cero excepto los 7 bits más significativos

Prototipo	Qué hace
<code>int toupper (int ch);</code>	Devuelve ch en mayúscula
<code>int tolower (int ch);</code>	Devuelve ch en minúscula

!!! OJO !!! Con las macros anteriores se debe evitar hacer operaciones como las siguientes :

```
x = isdigit (getch ());
y = isdigit (*p++);
```

Supongamos que la macro *isdigit()* está definida así :

```
#define isdigit(c) ((c) >= '0' && (c) <= '9')
```

Entonces las dos asignaciones anteriores se expandirían a :

```
x = ((getch ()) >= '0' && (getch ()) <= '9')
y = ((*p++) >= '0' && (*p++) <= '9')
```

El error cometido se ve claramente : en el primer caso nuestra intención era leer un sólo carácter y en realidad leemos dos, y en el segundo caso nuestra intención era incrementar en uno el puntero *p* y en realidad lo incrementamos en dos. Si *isdigit()* fuera una función en vez de una macro, no habría ningún problema.

## MATH.H

En esta librería se definen las funciones matemáticas.

```
int abs (int x);
Macro que devuelve el valor absoluto de un entero.
```

**double acos** (double x);  
Arcocoseno.

**double asin** (double x);  
Arcoseno.

**double atan** (double x);  
Arcotangente.

**double atof** (const char \*s);  
Convierte una cadena punto flotante.

**double cabs** (struct complex z);  
(TC) Valor absoluto de un número complejo.

**double ceil** (double x);  
Redondea por arriba.

**double cos** (double x);  
Coseno.

**double exp** (double x);  
Calcula e elevando a la x-xima potencia.

**double fabs** (double x);  
Valor absoluto de valor en punto flotante.

**double floor** (double x);  
Redondea por abajo.

**double fmod** (double x, double y);  
Calcula x módulo y, el resto de x/y.

**double frexp** (double x, int \*exp);  
Descompone un double en mantisa y exponente.

**long int labs** (long int x);  
Calcula el valor absoluto de un long.

**double log** (double x);  
Función logaritmo neperiano.

**double log10** (double x);  
Función logaritmo en base 10.

**double modf** (double x, double \*parte\_entera);  
Descompone en parte entera y parte fraccionaria.

**double pow** (double base, double exponente);

Función potencia, x elevado a y.

**double pow10** (int p);

(TC) Función potencia, 10 elevado a p.

**double sin** (double x);

Seno.

**double sqrt** (double x);

Calcula raíz cuadrada.

**double tan** (double x);

Tangente.

I

## STDIO.H

Es la librería estándar de entrada / salida. Las funciones aquí definidas hacen referencia a la entrada/salida de datos, ya sea a través de consola, pantalla o ficheros.

**int fclose** (FILE \*flujo);

Cierra un flujo abierto.

**int feof** (FILE \*flujo);

Macro que devuelve un valor distinto de cero si se ha detectado el fin de fichero en un flujo.

**int ferror** (FILE \*flujo);

Macro que devuelve un valor distinto de cero si ha ocurrido algún error en el flujo.

**int fgetc** (FILE \*flujo);

Obtiene un carácter de un flujo.

**int fgetpos** (FILE \*flujo, fpos\_t \*pos);

(TC) Obtiene la posición actual del puntero de fichero.

**char \*fgets** (char \*s, int n, FILE \*flujo);

Obtiene una cadena de caracteres de un flujo.

**FILE \*fopen** (const char \*nombre\_fichero, const char \*modo\_apertura);

Abre un flujo.

**int fputc** (int c, FILE \*flujo);  
Escribe un carácter en un flujo.

**int fputs** (const char \*s, FILE \*flujo);  
Escribe una cadena de caracteres en un flujo.

**int fread** (void \*buf, int tam, int n, FILE \*flujo);  
Lee datos de un flujo.

**int fseek** (FILE \*flujo, long desplazamiento, int origen);  
Posiciona el puntero de fichero de un flujo.

**long ftell** (FILE \*flujo);  
Devuelve la posición actual del puntero de fichero.

**int fwrite** (const void \*buf, int tam, int n, FILE \*flujo);  
Escribe en un flujo.

**int getc** (FILE \*flujo);  
Macro que obtiene un carácter de un flujo.

**int getchar** (void);  
Macro que obtiene un carácter de *stdin*.

**char \*gets** (char \*string);  
Obtiene una cadena de caracteres de *stdin*.

**void perror** (const char \*s);  
Mensajes de error del sistema.

**int putc** (int c, FILE \*flujo);  
Escribe un carácter en un flujo.

**int putchar** (int c);  
Escribe un carácter en *stdout*.

**int puts** (const char \*s);  
Escribe un string en *stdout* (y añade un carácter de nueva línea).

**int remove** (const char \*nombre\_fichero);  
Función que borra un fichero.

**int rename** (const char \*viejo\_nombre, const char \*nuevo\_nombre);  
Renombra un fichero.

**void rewind** (FILE \*flujo);  
Reposiciona el puntero de fichero al comienzo del flujo.

**int sprintf** (char \*buffer, const char \*formato [, argumento, ...]);  
Envía salida formateada a un string.

**int sscanf** (const char \*buffer, const char \*formato [, direccion, ...]);  
Ejecuta entrada formateada de string.

**int ungetc** (int c, FILE \*flujo);  
Devuelve un carácter al flujo de entrada.

## STDLIB.H

Es la librería estándar del C. En ella se definen las funciones de uso más común.

**void abort** (void);  
Termina anormalmente un programa.

**double atof** (const char \*s);  
Convierte una cadena a un punto flotante.

**int atoi** (const char \*s);  
Convierte una cadena en un entero.

**long atol** (const char \*s);  
Convierte un string a un long.

**div\_t div** (int numer, int denom);  
Divide dos enteros.

**char \*ecvt** (double valor, int ndig, int \*dec, int \*sign);  
(TC) Convierte número en coma flotante a cadena.

**void exit** (int estado);  
Termina programa.

**char \*gcvt** (double valor, int ndec, char \*buf);  
(TC) Convierte un número en coma flotante a string.

**char \*itoa** (int valor, char \*cad, int radix);  
Convierte un entero a una cadena.

**char \*ltoa** (long valor, char \*cadena, int radix);  
Convierte un long en una cadena.

**max(a,b)**

(TC) Macro que genera código en línea para encontrar el valor máximo de dos enteros.

**min(a,b)**

(TC) Macro que genera código en línea para encontrar el valor mínimo de dos enteros.

**int rand (void);**

Generador de números aleatorios.

**int random (int num);**

(TC) Macro que devuelve un entero.

**void randomize (void);**

(TC) Macro que inicializa el generador de números aleatorios.

**double strtod (const char \*inic, char \*\*fin);**

Convierte cadena a double.

**int system (const char \*comando);**

Ejecuta un comando DOS.

**char \*ultoa (unsigned long valor, char \*cadena, int radix);**

(TC) Convierte un unsigned long a una cadena.

## STRING.H

En esta librería se definen las funciones de manejo de cadenas.

**char \*strcat (char \*destino, const char \*fuente);**

Añade fuente a destino.

**char \*strchr (const char \*s, int c);**

Encuentra c en s.

**int strcmp (const char \*s1, const char \*s2);**

Compara un string con otro.

**char \*strcpy (char \*destino, const char \*fuente);**

Copia el string fuente al string destino.

**char \*\_strerror (const char \*s);**

(TC) Construye un mensaje de error hecho a medida.

**char \*strerror** (int numerr);

Devuelve un puntero al string que contiene el mensaje de error.

**size\_t strlen** (const char \*s);

Calcula la longitud de un string.

## ALLOC.H (TC)

En esta librería se definen las funciones relacionadas con la asignación dinámica de memoria.

**void \*calloc** (size\_t nelems, size\_t tam);

Asigna memoria principal.

**unsigned coreleft** (void);

Devuelve la cantidad de memoria no usada. Modelos tiny, small, y medium.

**unsigned long coreleft** (void);

Devuelve la cantidad de memoria no usada. Modelos compact, large, y huge.

**void free** (void \*bloque);

Libera bloques asignados con *malloc()* o *calloc()*.

**void \*malloc** (size\_t tam);

Asigna memoria principal.

**void \*realloc** (void \*bloque, size\_t tam);

Reasigna memoria principal.

## BIOS.H (TC)

En esta librería se definen las funciones relacionadas con la ROM BIOS.

**int bioscom** (int cmd, char byte, int puerto);

E/S de comunicaciones RS-232

**int biosprint** (int cmd, int byte, int puerto);

E/S de impresora usando directamente la BIOS.



## CONIO.H (TC)

En esta librería se definen las principales funciones de entrada/salida por consola.

**void clrscr (void);**

Borra ventana de texto.

**int cprintf (const char \*formato [, argumento,...]);**

Escribe salida formateada en la ventana de texto en la pantalla.

**int scanf (char \*formato [ , direccion, ... ]);**

Lee entrada formateada de consola.

**int getch (void);**

Lee un carácter de la consola sin eco en pantalla.

**int getche (void);**

Lee un carácter de la consola, con eco en pantalla

**void gotoxy (int x, int y);**

Posiciona cursor en ventana de texto.

**void highvideo (void);**

Selecciona caracteres de texto en alta intensidad.

**int kbhit (void);**

Chequea para ver si se ha pulsado alguna tecla. es decir, para ver si hay alguna tecla disponible en el buffer de teclas.

**void lowvideo (void);**

Selecciona salida de caracteres en ventana de texto en baja intensidad.

**void normvideo (void);**

Selecciona caracteres en intensidad normal.

**int putch (int ch);**

Escribe un carácter en la ventana de texto sobre en la pantalla.

**void textbackground (int nuevocolor);**

Selecciona nuevo color de fondo de los caracteres en modo texto.

**void textcolor (int nuevocolor);**

Selecciona nuevo color de texto de los caracteres en modo texto.

**void textmode (int nuevomodo);**

Cambia modo de pantalla (en modo texto).

int **ungetch** (int ch);

Devuelve un carácter al teclado.

int **wherex** (void);

Devuelve posición horizontal del cursor dentro de la ventana de texto corriente.

int **wherey** (void);

Devuelve posición vertical del cursor dentro de la ventana de texto corriente.

void **window** (int izq, int ar, int der, int ab);

Define ventana activa en modo texto.

## DIR.H<sup>1</sup> (TC)

Es esta librería se definen las funciones de manejo de directorios.

int **findfirst** (const char \*nombrepath, struct fblk \*ffblk, int atributo);

Busca un directorio en el disco.

int **findnext** (struct fblk \*ffblk);

Continúa la búsqueda iniciada por *findfirst()*.

int **getcurdir** (int unidad, char \*directorio);

Obtiene directorio actual para la unidad especificada.

char \***getcwd** (char \*buffer, int longitud\_buffer);

Obtiene directorio de trabajo actual.

int **getdisk** (void);

Obtiene unidad actual.

int **mkdir** (const char \*path);

Crea un directorio.

int **rmdir** (const char \*path);

Quita un directorio.

int **setdisk** (int unidad);

Pone la unidad de disco actual.

## DOS.H (TC)

En esta librería se definen las funciones relacionadas con el Sistema Operativo MS-DOS.

**void ctrlbrk** (int (\*manejador) (void));

Pone manejador de control-break.

**void delay** (unsigned milisegundos);

Suspende la ejecución durante un intervalo (en milisegundos).

**void disable** (void);

Inhabilita las interrupciones.

**void enable** (void);

Habilita las interrupciones hardware.

**int getcbk** (void);

Obtiene el estado de control-break.

**void getdfree** (unsigned char drive, struct dfree \*dtable);

Obtiene el espacio libre de disco.

**unsigned char inportb** (int portid);

Lee un byte de un puerto hardware.

**int int86** (int intno, union REGS \*inregs, union REGS \*outregs);

Interrupción de software 8086.

**void keep** (unsigned char estado, unsigned tamaño);

Termina y queda residente.

**void nosound** (void);

Desactiva el altavoz del PC.

**void outportb** (int portid, unsigned char value);

Escribe un byte en un puerto hardware.

**int setcbk** (int valorcbk);

Pone el estado de control-break.

**void sound** (unsigned frecuencia);

Activa el altavoz del PC a una frecuencia especificada.

**int unlink** (const char \*nombre\_de\_fichero);

Borra un fichero.



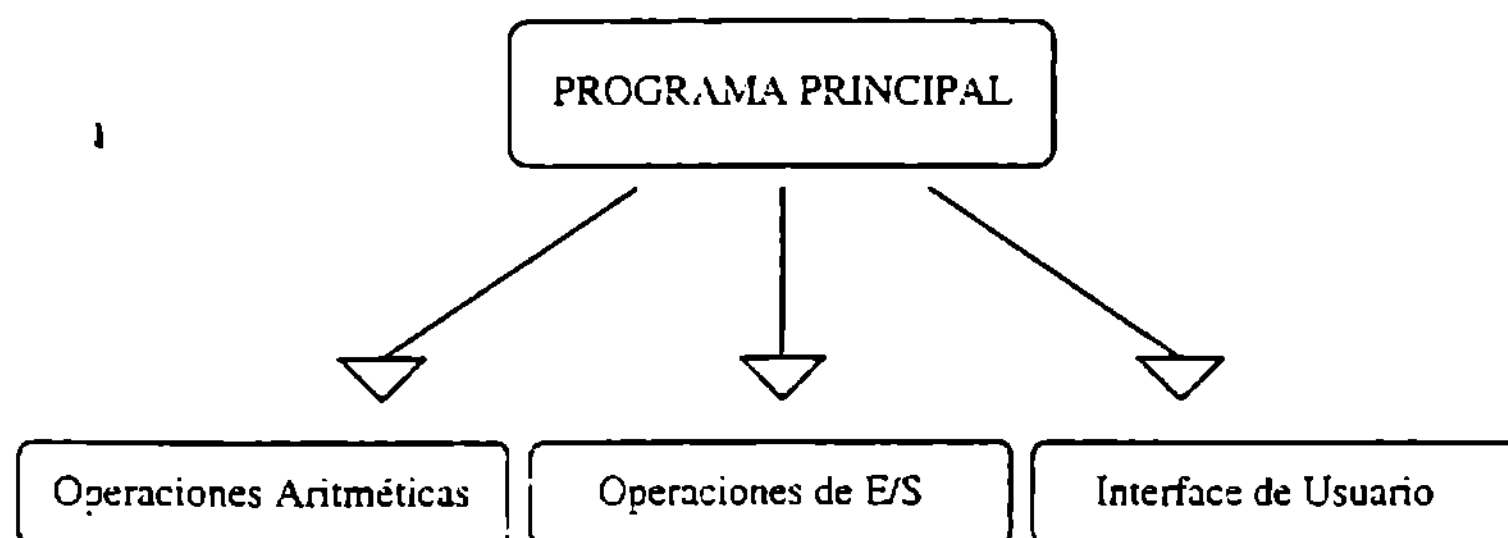
## *Ficheros Proyecto*

Cuando diseñamos grandes aplicaciones se hace necesario estructurar nuestros programas, dividiéndolos en **módulos**. Cada uno de estos módulos contendrá una serie de funciones, constantes, variables, definiciones de estructuras, etc; todas ellas tienen en común que sirven para resolver un cierto subproblema de nuestro problema global. En la mayoría de los lenguajes modernos esta forma de trabajo se puede llevar a cabo con distintos nombres, pero con la misma finalidad., dando lugar a lo que se conoce como **programación modular**.

En C cada uno de los módulos en los que dividimos un programa dará lugar, en general, a dos ficheros. En el lenguaje Pascal, cada módulo da lugar a un único fichero, llamado **unidad**, que además recibe un tratamiento especial. Pero la creación de módulos en C no es tan sencilla como pueda ser en Pascal. Así, en el siguiente punto trataremos de ver dónde radican esas *dificultades*.

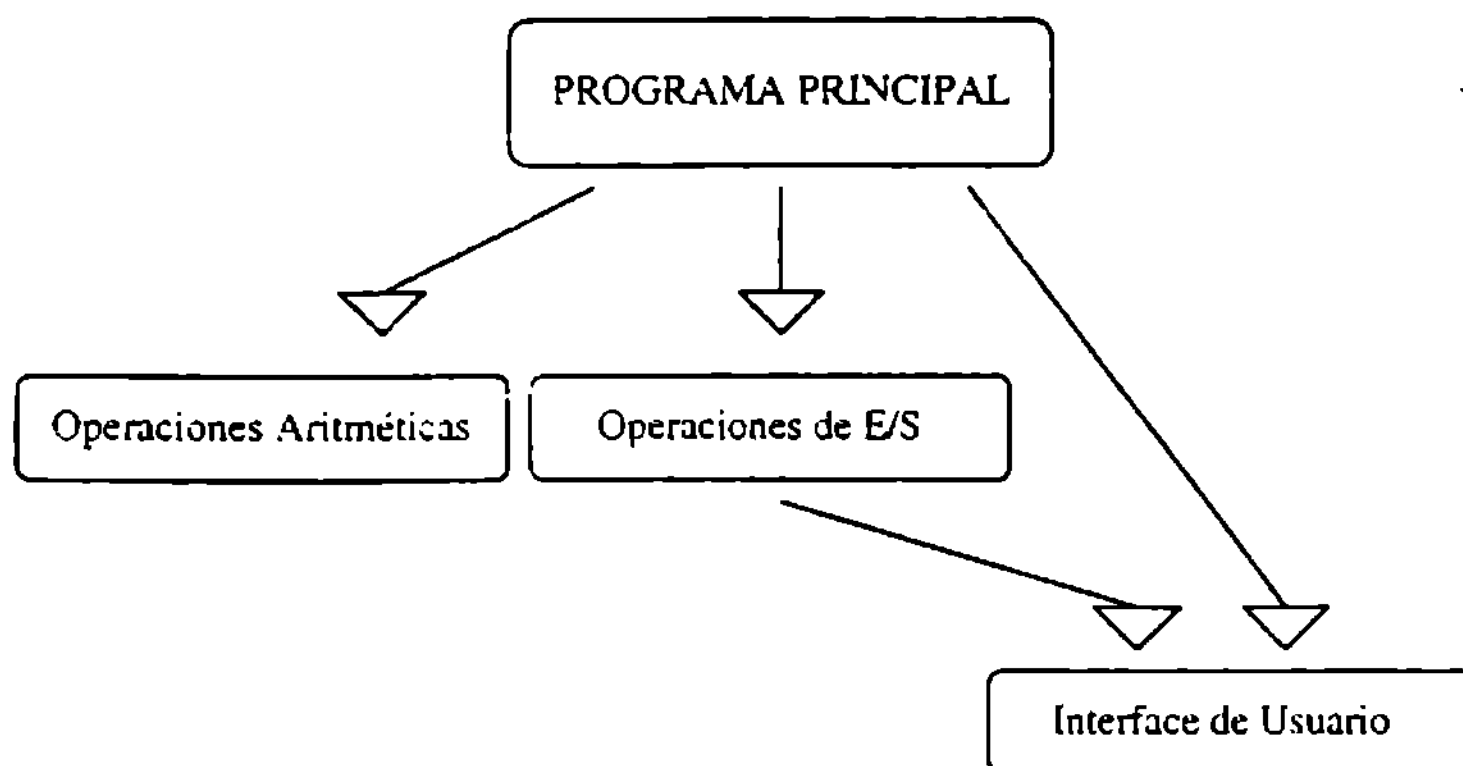
Hemos dicho que las grandes aplicaciones las vamos a dividir en módulos. De esta manera, si tenemos que construir una calculadora, podríamos dividir nuestro problema en, al menos, tres módulos : (Figura C.1)

1. En el primero introducimos todo lo relacionado con las operaciones aritméticas.
2. En el segundo pondremos todo lo relacionado con la entrada/salida de datos.
3. Finalmente, en el tercero irá todo lo relacionado con el interface de usuario.
4. También necesitaremos un programa principal, que haga las llamadas a función necesarias para que la calculadora funcione correctamente.



**Figura C.1.** Esquema de programa modular para una calculadora.

Hasta ahora todo parece muy sencillo. La cosa se complica cuando, por ejemplo, alguna de las operaciones de entrada/salida necesita de alguna de las operaciones del interface de usuario, o viceversa. Generalizando, cuando para poder implementar operaciones de un módulo se necesita de operaciones de otros módulos. (Figura C.2)



**Figura C.2.** Otro esquema de programa modular para una calculadora.

En Pascal una situación como ésta no presenta ningún inconveniente. Pero en C sí que vamos a tener problemas. Para entenderlo seguiremos con el ejemplo de la calculadora. Recordando el capítulo dedicado a las directivas de compilación, para poder acceder a las operaciones de otro fichero, debemos *incluirlo*. Para ello tenemos la directiva de inclusión (`#include <nombre_fichero>`). El preprocesador sustituye esta directiva por el fichero especificado, de tal forma que el resultado práctico es que en el fichero preprocesado se copia el fichero especificado en el lugar ocupado por la directiva `#include`. (Figura C.3)

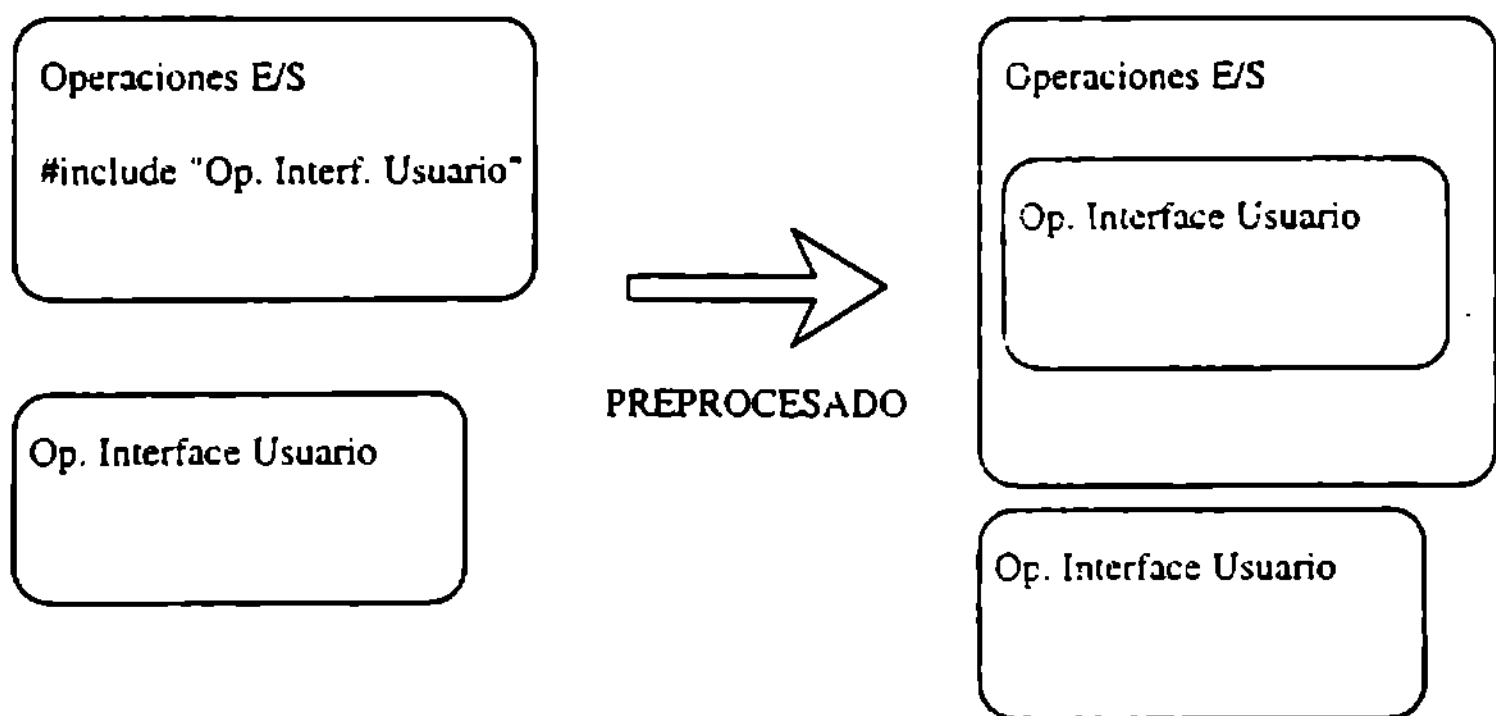


Figura C.3. Efecto del preprocesado de una directiva de inclusión.

De esta forma, todas las operaciones definidas dentro del fichero de Interface de Usuario son visibles para el fichero de Operaciones de E/S, y por tanto, ya pueden usarse. El problema se presenta cuando a continuación se realiza el preprocesado del fichero que contiene el programa principal. Como también incluye el fichero de Interface de Usuario, éste aparecerá duplicado (Figura C.4). Ahora cuando el compilador actúe se encontrará código repetido y dará los mensajes de error correspondientes.

## Ficheros Proyecto.-

Para resolver este problema recurrimos a los llamados **ficheros proyecto**, que son ficheros especiales, que no contienen código C, sino una lista con los ficheros que componen la aplicación, de forma que si el compilador encuentra algo repetido sabe cómo actuar.

Pero antes de ver cómo se construyen vamos a ver otro aspecto importante. Al comenzar este apéndice decíamos que en C cada módulo daba lugar generalmente a dos ficheros. ¿Qué contiene cada uno de ellos? El primero contiene la *declaración* de las operaciones del módulo, mientras el segundo contiene la *implementación* de las mismas.

En Pascal todo ello va en un sólo fichero (unidad). A continuación veremos la estructura de cada uno de ellos.

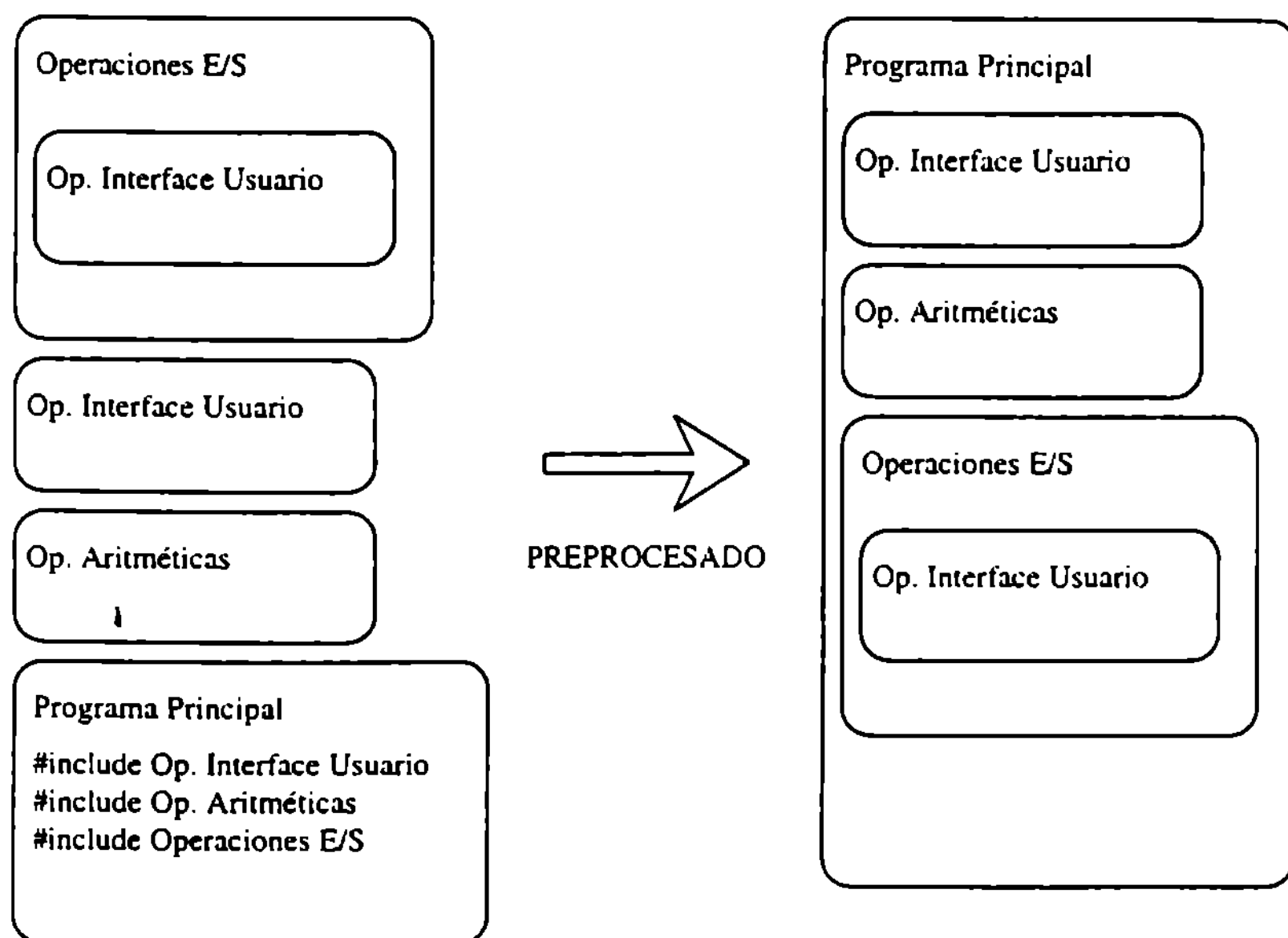


Figura C.4. Efecto del preprocesado del programa principal.

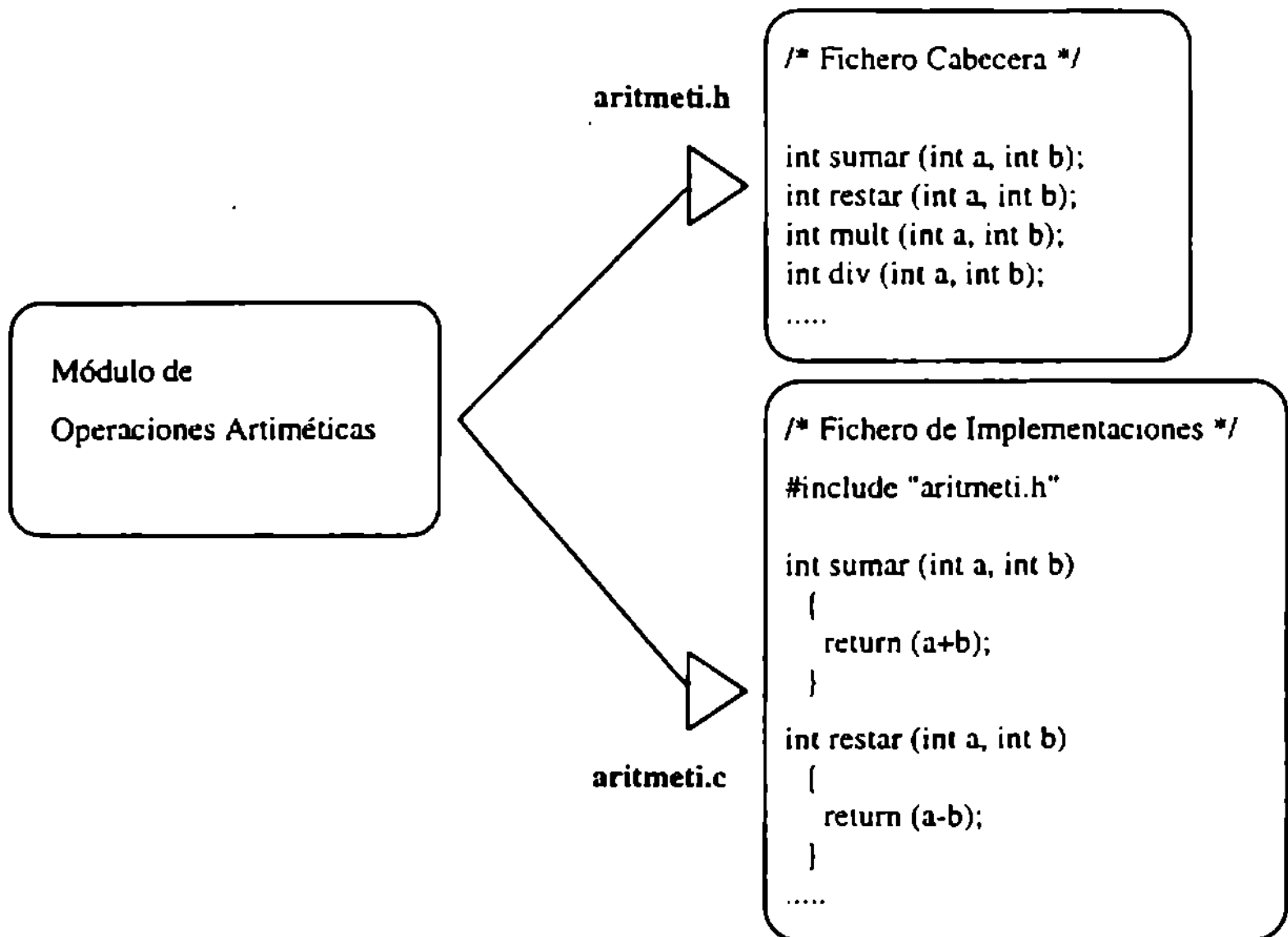
El fichero que contiene las declaraciones se llama **fichero cabecera**, y suele nombrarse igual que el fichero de implementaciones pero con extensión **".H"**. En este fichero debemos poner la declaración de las funciones que se consideran *visibles* fuera del módulo, así como las constantes, variables o estructuras que queremos que se puedan usar fuera del mismo. Lo que nunca debe aparecer es código de alguna función o cualquier otra cosa que sea interna al módulo que estamos diseñando.

El otro fichero se llama **fichero de implementación**, y como su nombre indica contiene las implementaciones relativas a todas las declaraciones especificadas en el fichero cabecera, así como todas las cosas que se consideren internas al módulo, es decir, visibles únicamente dentro del mismo.

En la figura C.5 se muestra un ejemplo del posible contenido de los ficheros cabecera y de implementación para el módulo de operaciones aritméticas.

De esta manera, para definir un fichero proyecto sólo tenemos que incluir en él los ficheros de implementaciones, es decir, sólo se especifican los ficheros que contienen el código de las operaciones que necesitamos de cada uno de los módulos que componen la aplicación. Sin embargo, dentro de los ficheros de implementación debe aparecer una directiva de inclusión referida a su fichero cabecera correspondiente.





**Figura C.5.** Ficheros a los que da lugar el módulo de Operaciones Aritméticas.

Hasta ahora no hemos visto nada relacionado con la forma que tiene un fichero proyecto. Decíamos que no contenía código C, sino el nombre de los ficheros que componen la aplicación (sólo los **ficheros con implementaciones**): Por ejemplo, en el caso de la calculadora de la que hemos estado hablando, su fichero proyecto contendría el nombre de los ficheros de implementación correspondiente a los módulos de operaciones aritméticas, de entrada/salida y de interface de usuario, así como el nombre del fichero que contiene el programa principal.

El nombre del fichero proyecto debe tener extensión **".PRJ"**. A partir de ahora el fichero a compilar es el propio fichero proyecto, para que el compilador tenga en cuenta el problema visto de la duplicación de código. El nombre del fichero ejecutable será el mismo del fichero proyecto pero con extensión **".EXE"**.

Una cosa muy importante a tener en cuenta es que los distintos módulos deben poder *compilarse* de forma independiente, aunque *no linkarse*. Es decir, como en C el proceso de compilación se realiza completo antes de que actúe el *linkador*, cada módulo debe compilar correctamente, ya que sintáctica y semánticamente será correcto; pero sólo tendrá sentido cuando las referencias a otros módulos sean conocidas, y eso ocurre después de que el *linkador* recomponga todo el programa.

## Ficheros proyecto en Borland C++.-

Para construir un fichero proyecto debemos tener muy presente cuál es el compilador que vamos a usar, ya que según la marca y la versión se hace de forma diferente. Nosotros vamos a centrarnos en el compilador de Borland C++ versión 3.1.

En cualquier caso, para el resto de compiladores la idea básica es similar, pero la forma de llevarla a cabo cambia. Así en la versión de Turbo C 2.0, el fichero proyecto es un fichero de texto que se puede escribir desde cualquier editor, y que contiene exclusivamente los nombres de los ficheros que componen la aplicación.

Sin embargo, para la versión de compilador elegida se construye dentro del entorno de desarrollo, como una ventana especial de trabajo. Dentro del menú principal existe una opción ("*Open project*") para generar ficheros proyecto (figura C.6).

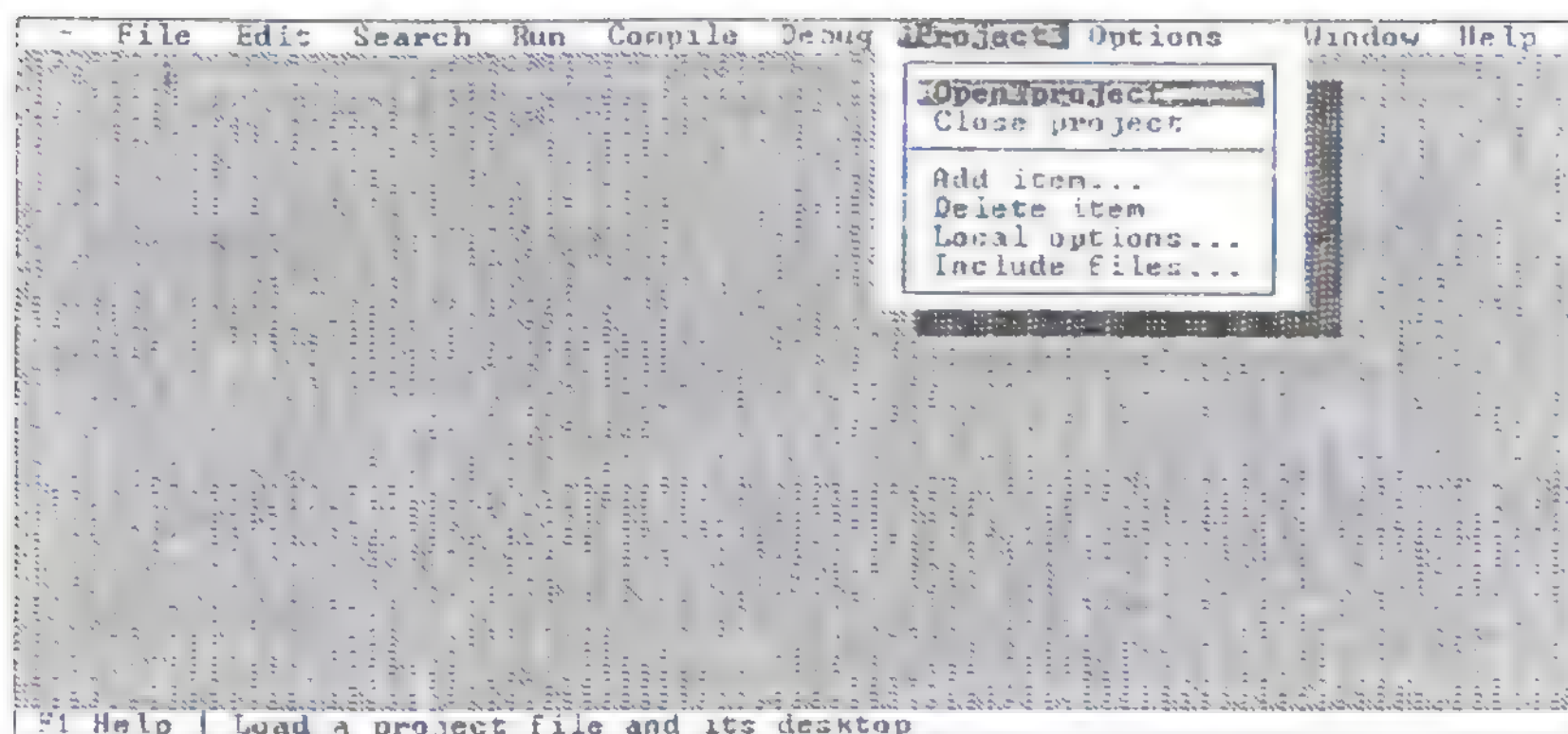


Figura C.6. Opción "Project" del menú principal de Borland C++ v3.1.

Al entrar en esta opción nos aparece un nuevo submenú, con todo lo necesario para crear y modificar ficheros proyecto. Así, si entramos en la opción de apertura de proyectos, nos aparecerá una ventana similar a la de apertura de ficheros, donde podemos localizar nuestro fichero ".prj". Si no estaba creado, al poner el nombre del nuevo fichero se creará, apareciendo una ventana de proyectos como la de la figura C.7.



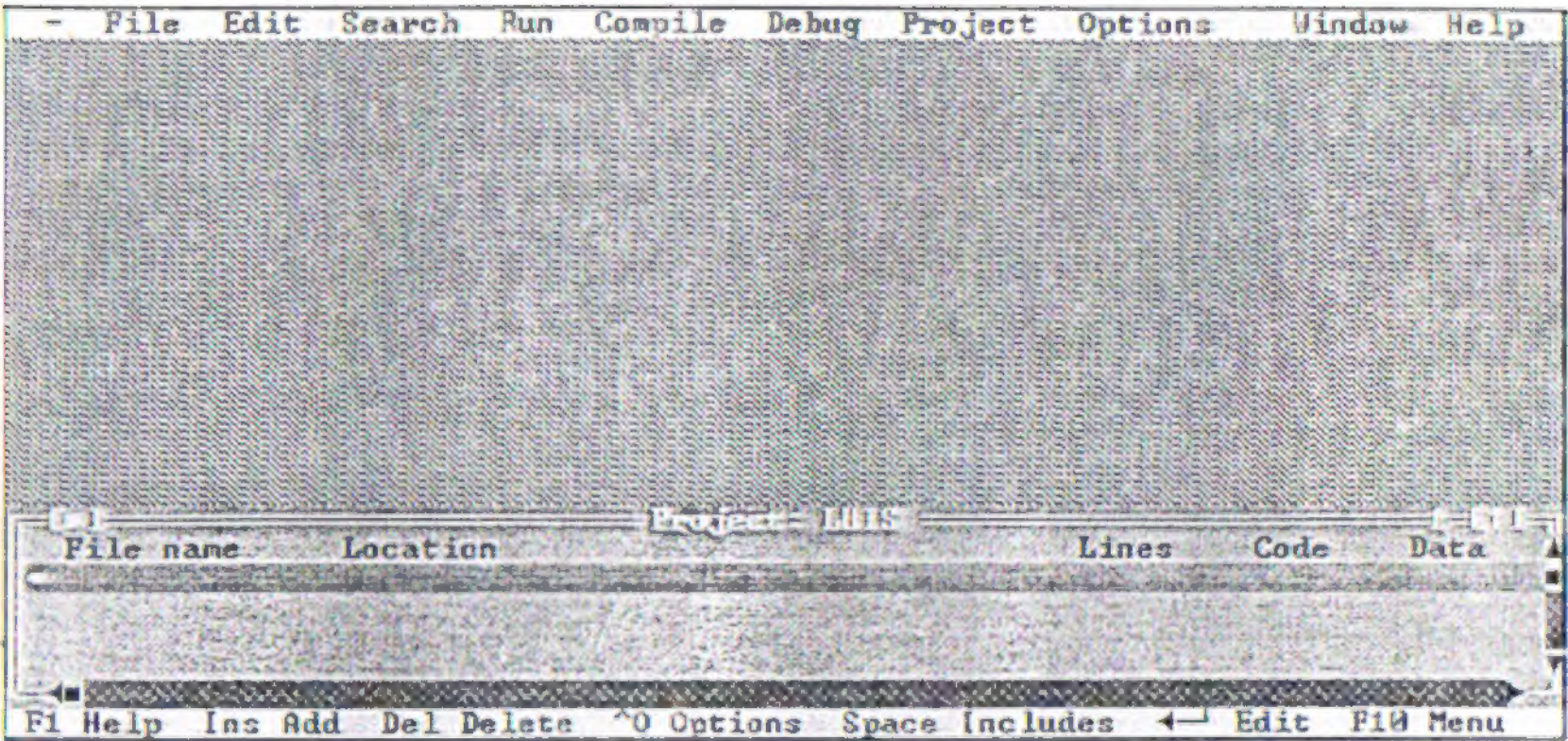


Figura C.7. Ventana de Fichero Proyecto de Borland C++ v3.1.

Ahora sólo nos falta rellenar esta ventana con los nombres de los ficheros que componen la aplicación, que recordamos son los ficheros de implementación más el fichero que contiene el programa principal. Para ello no hace falta escribir los nombres, sino entrar en la opción “Add item” del submenú (figura C.6). Se abrirá una ventana en la que podemos seleccionar los ficheros. (Figura C.8).

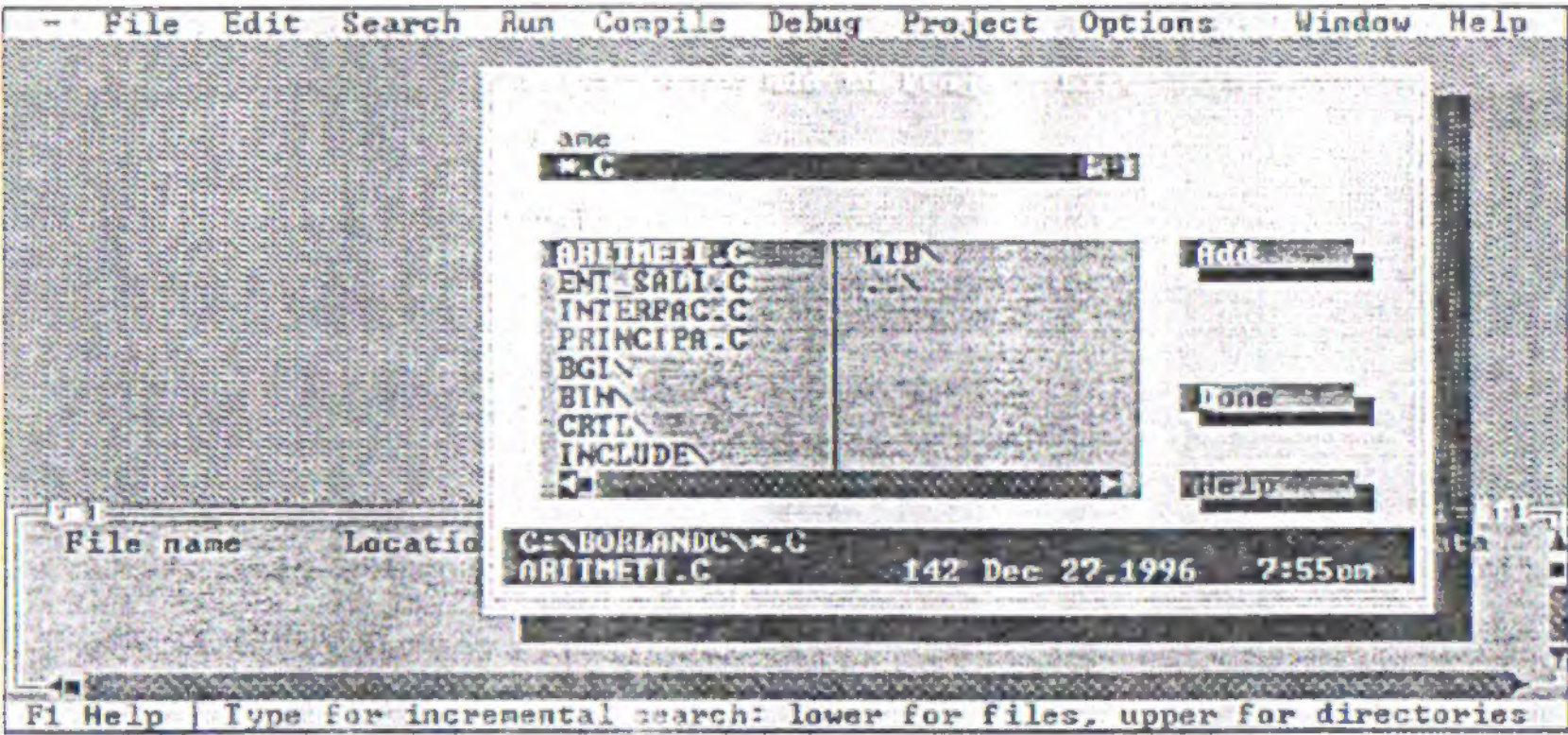


Figura C.8. Ventana de selección de componentes del Fichero Proyecto.

Seleccionamos uno a uno los ficheros. Cada vez que marquemos uno aparecerá su nombre en la ventana de proyecto, y se volverá a abrir la ventana de selección para poder añadir más ficheros. Cuando estén todos, se pulsa el botón “Done”. Ahora ya podemos compilar nuestro programa completo. Para ello, estando activa la ventana que acabamos de construir, damos la orden de compilar.



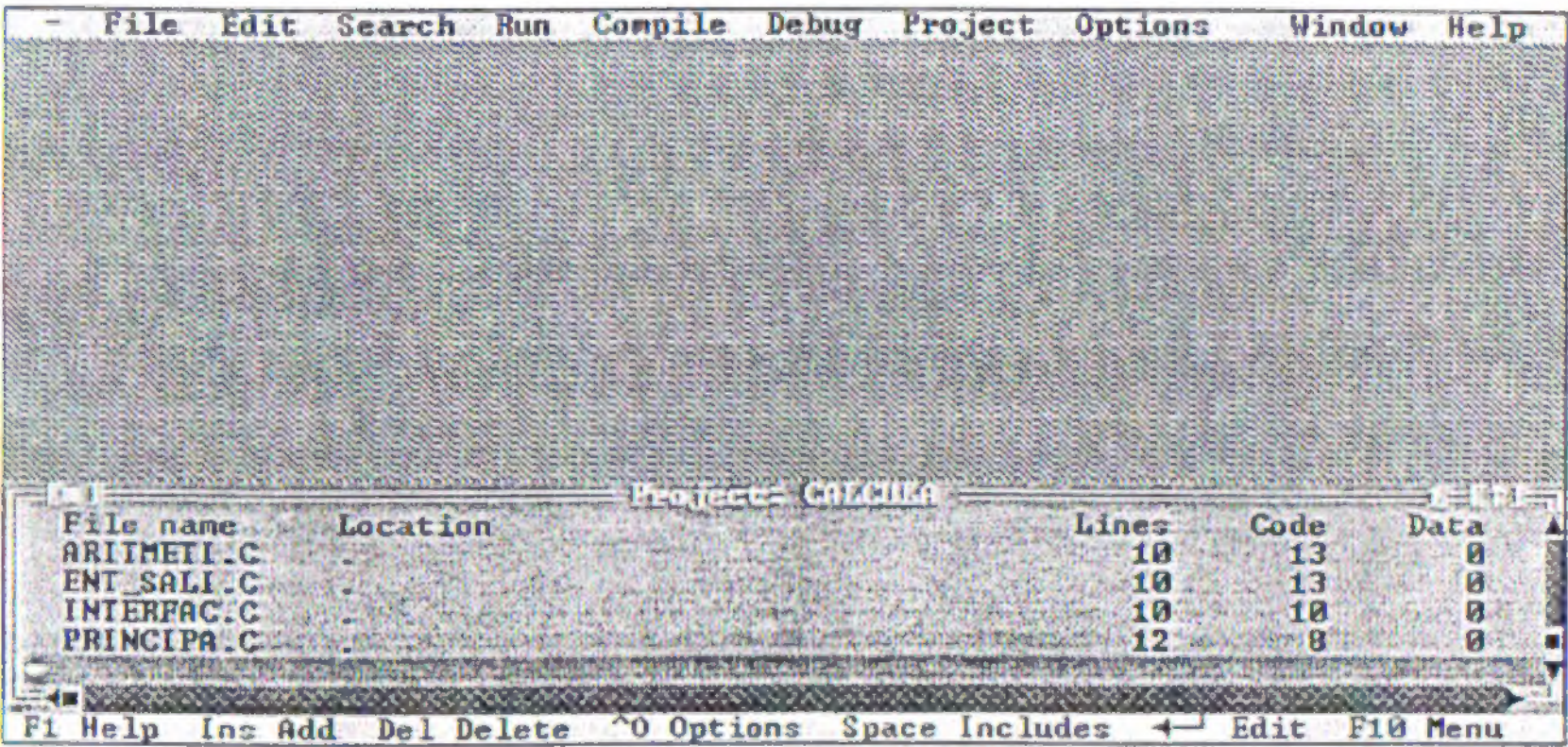


Figura C.9. Ventana de Proyecto después de compilar.

Se generará el fichero ejecutable correspondiente. La figura C.9 representa el estado de la ventana de proyecto una vez compilado el programa. A la derecha de la misma aparecen las líneas compiladas de cada fichero. Como puede apreciarse, no es necesario que los ficheros de la aplicación estén abiertos, basta con abrir la ventana de proyecto.

## *Bibliografía recomendada*



Programación en Turbo C. Segunda edición.  
Herbert Schildt.  
Editorial Borland-Osborne / McGraw-Hill 1991.



Libro especialmente recomendado para iniciarse en la programación en C.



El lenguaje de programación C.  
Brian Kernighan & Dennis Ritchie  
Editorial Prentice-Hall. Versión traducida de 1991.



Escrito por los creadores del lenguaje, es una referencia básica para cualquier programador de C.



Gestión de entrada / salida en C.  
Salvador Senent Díez y Rosa Domínguez Gómez.  
Editorial Anaya Multimedia 1992.



Es una amplia guía dedicada exclusivamente a la entrada/salida.



Lenguaje C. Introducción a la programación.  
Kelley / Pohl.  
Editorial Addison-Wesley 1987.



Contiene muchos ejemplos para el C de UNIX.



La biblia del Turbo C : fundamentos y técnicas avanzadas de programación.  
Scott Zimmerman  
Editorial Anaya Multimedia 1989.



Completo manual de programación en C, tanto a nivel básico como avanzado.



Turbo C/C++.  
Herbert Schildt.  
Editorial Osborne / McGraw-Hill 1992



Este libro servirá de gran ayuda a quien decida migrar de C a C++.



Design Patterns  
Elements of Reusable Object-Oriented Software.  
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.  
Editorial Addison Wesley Sixth Printing, April 1996.



Libro avanzado de programación en C++.



Estructuras de Datos, Algoritmos, y Programación Orientada a Objetos.  
Gregory L. Heileman.  
Editorial McGraw-Hill 1998



Libro para la implementación de Estructuras de Datos Avanzadas en C++.